(4)

STRUCTURE BASED FORMAL METHODS FOR SOFTWARE ENGINEERING

A002: **Final Report**
Report issued July 27, 1989

Prepared by:

Mark Moriconi (Principal Investigator)
Computer Science Laboratory
SRI International, Menlo Park CA 94025

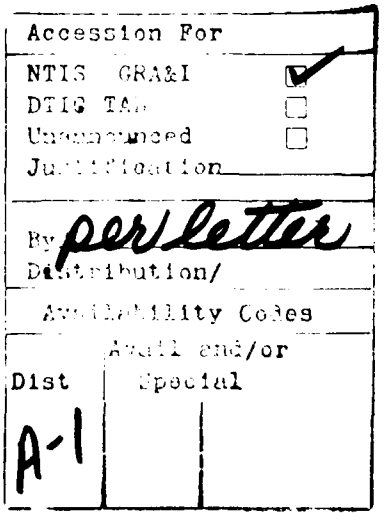
Prepared for:

Dr. Ralph Wachter
Computer Sciences Division, Code 1133
Office of Naval Research
Department of the Navy
800 North Quincy Street
Arlington, VA 22217-5000
cc: Mr. Philip A. Harless, Contracting Officer (Code S0507A)
Director, Naval Research Laboratory
Attn: Code 2627, Washington, DC 20375

Contract No. N00014-83-C-0300
SRI Project 5912

SRI International
333 Ravenswood Ave. • Menlo Park, CA 94025
415 326 6200 • TWX 910-373-2046 • Telex 334 486

Contents



1 Motivation for the Research

It is widely recognized that the cost of software is far outstripping that of hardware, and that software repair, improvement, and enhancement typically consume the major portion of the cost. This is true even in light of recent advances in languages and tools, which are more than offset by the increasing size and complexity of software systems.

There are several reasons for the high cost of modifications. One is that system requirements may be wrong or imprecise. Research on rapid prototyping is expected to help in this regard. Another reason is that a system implementation may not meet its requirements. Approaches to this problem include testing, formal verification, and runtime assertion checking.

However, the dominant source of cost is system modification, which is necessitated by changes in requirements and in the support environment. If the design or the implementation of a large system is changed, the incremental cost of an individual change can be unacceptably high because a seemingly minor change to one part of a system can have unforeseen and subtle consequences in another part.

Some changes always will have far-reaching effects. The root cause is the implementation goal of good performance which usually dominates and conflicts with the goals of maintaining clarity and structure. In this research, we have devised formal techniques that can substantially reduce the cost of modifying large systems, especially those systems that have been optimized for performance. The techniques have been implemented and apply to a large class of sequential systems containing such objects as modules, procedures, and variables.

The ways in which objects can be related is limited at present, reducing design and implementation flexibility. We believe that our results can be generalized to handle powerful parameterization mechanisms, object-oriented paradigms, and concurrency. However, this has not yet been done.

2 Our Basic Approach

A formal system development involves the two transitions shown in Figure 1. First, an informal description of requirements is transformed into a formal statement of *what* the sytem is intended to do. Second, that formal specification is transformed into an implementation that describes *how* to perform the specified computation. It is the what-to-how transformation that is the subject of this research. Henceforth, we use the terms "structural design" and "design" to refer to what-to-how transformations.

For managing the evolution of a system, the crucial difference between a specification and an implementation is not the difference in concreteness. It is the complexity

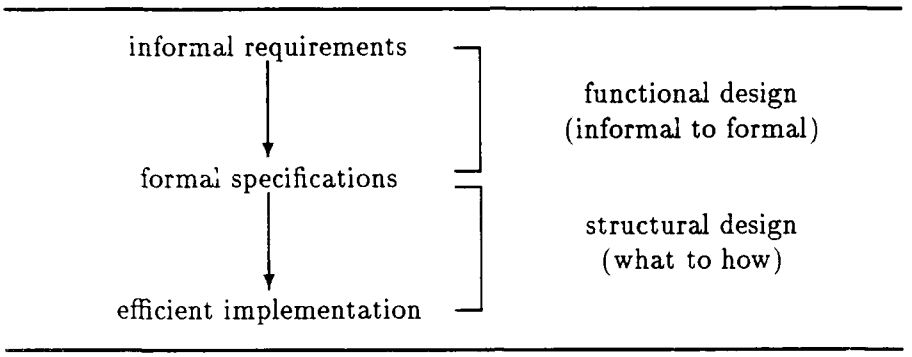


Figure 1: Steps in a formal system development

of the interconnections among system objects. An implementation, especially an efficient one, will invariably be highly interconnected and somewhat unstructured. On the other hand, performance is not an issue in specifications and, hence, they tend to be modular.

Therefore, our approach to change focuses on the formal documentation and analysis of large-system structures. In particular, we formally record the structural design of a system during its development, and then use the record to:

1. **Explain system organization**. A formal record of structural design decisions can explain how abstract objects and interactions are evolved into a concrete implementation. This information is needed to make changes to a system.

2. **Control implementation connectivity**. Structural invariants can be enforced automatically. It is decidable whether or not a structural design and its implementation are consistent under certain reasonable assumptions.

3. **Find the effects of changes.** If the structural design or implementation of a system is changed, new bugs can be introduced. The number of new bugs can be greatly reduced if developers are provided with an accurate assessment of the effects of a planned change. The semantic effects of a change can be isolated in a system design or implementation through an analysis of (i.e., proofs about) its structure.

3 Summary of Main Results

Our research has resulted in four major results:

1. The first formal technique for specifying implementation structures. Existing formal specification languages can describe the structure of a specification, but

not the intended structure of an implementation. A specification of implementation structure is needed to explain and enforce structural design decisions, the key to managing complexity in large systems.

2. A new method for reasoning about changes to a system. The problem of reasoning about changes was originally formalized by this author in his Ph.D. thesis [11]. That solution used a Hoare-style logic and had the practical drawback of undecidability. We have devised a radically new approach to the problem that involves a structural approximation of the semantic effects of a change. An approximate solution is intuitively appealing and it can be found in a decidable theory.

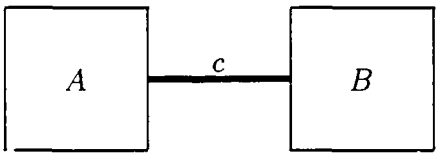
3. A new technique for controlling interconnections in a system implementation using standard program analysis and formal verification technology. The question of whether a structural specification is consistent with an implementation is decidable under certain reasonable assumptions.

4. An innovative prototype system — called PegaSys — that uses pictures as formal documentation. To our knowledge, PegaSys is the first system to manipulate nontrivial design structures in ways that take into account their semantics. There are over a dozen commercially available interactive diagramming systems that have proved successful in industry but all lack semantics.

Two papers are attached to this final report as appendices. They explain the points above in some detail. The remainder of this section summarizes our results in the three areas.

3.1 Structural Designs

To illustrate what is meant by structural design, consider the following diagram:



A superficial reading of this diagram might be that boxes A and B are abstract operations and the bold line is a communication channel c between them. Although this reading gives a general idea of what is meant by the diagram, it is much too imprecise for our purposes. Here are some properties the diagram might indicate:

- **Direct communication**: Operations A and B communicate directly through channel c when they are executed.

- **Indirect communication**: Operations A and B communicate indirectly through intermediaries via channel c. Indirect relationships result from the cumulative effects of procedure calls on nonlocal objects and on objects passed as parameters.

- **Completeness**: The only way that A and B communicate is through channel c. That is, there are no other channels between A and B.

A formal description of the above property is needed so that there is no ambiguity about what is intended and so that analysis based on the specification will be meaningful.

The formal structural specification of a system consists of multiple levels of detail, each level containing abstractions appropriate to that level. Most abstract objects are represented naturally as primitive procedures or variables which are subsequently implemented in terms of one or more similar objects. On the other hand, most abstract connections are expressed as derived concepts defined in terms of more primitive connections. Abstract connections are used to partition a system into manageable parts that interact in well-defined and predictable ways.

Next, we present three concrete examples of structural design concepts, defined in terms of the following primitive concepts of our logic:

$mod(P, x)$ means that procedure P modifies variable x directly or indirectly through a called procedure. The *mod* relation is used for global program optimization in compilers.

$x \stackrel{P}{\Longrightarrow} y$ means that information flows from a variable x to a variable y under procedure P provided a change in the value of x can be conveyed to y when P is executed. For example, the binding of an actual parameter a to a formal parameter x causes a flow from a to x.[1]

$\Longrightarrow_f$, $\Longrightarrow_b$, $\Longrightarrow_l$ stand for forward, backward, and lateral information flow, respectively. Forward and backward flows model the interprocedural variable bindings that result from a direct or transitive procedure call. Lateral flow is intraprocedural, involving local variables of the same procedure. Henceforth, $x \Longrightarrow y$ is taken to mean that $\langle x, y \rangle$ is a forward, backward, or lateral flow.

[1]Classical information theory, Shannon [13] and others, is concerned with the *amount* of information generated by a particular event. We are interested in the simpler qualitative question of whether *any* information is generated by an event. In other words, we are interested in whether a change affects an object at all, not in how much it affects it.

callByVR$(P, Q, plist)$ models a procedure call from procedure P directly to procedure Q with an arbitrary number of actual-formal parameter pairs (*plist*) having a value-result semantics.

Types *var* and *proc* are used to denote variables and procedures, respectively. Variables of type *vvar* are used to specify the different value assignments to an ordinary variable. The predicate *versionOf(x,y)* tells whether a version variable x is associated with a variable y and the predicate *varOf(x,P)* tells whether version variable x is associated with procedure P.

Example 1 *Protecting a variable.* It is often useful to restrict access to a variable or to restrict the ways in which a variable can be used. For instance, we may want to allow procedures to read a certain variable but not allow them to write it. This is easily formalized using the predicate

$$\text{ReadOnly: var} \times \text{proc} \rightarrow \text{bool}$$

which is defined by

$$\text{ReadOnly}(x, P) \stackrel{\text{def}}{=} \neg\text{mod}(P, x)$$

for x in *var* and P in *proc*. For a given variable v, we can write

$$(\forall p\text{: proc})\text{ReadOnly}(v, p)$$

to indicate that no procedure p can modify v. □

Example 2 *Restricting variable interactions.* A set of variables can be partitioned into independent subsets using a predicate which says that a variable x is completely independent of a variable y if and only if a change in the value of y has no effect on the value of x. This predicate

$$\text{IndependentOf: var} \times \text{var} \rightarrow \text{bool}$$

is defined, for x and y in *var*, by

$$\text{IndependentOf}(x, y) \stackrel{\text{def}}{=}$$
$$(\forall \hat{x}, \hat{y}\text{: vvar})(\forall R\text{: proc})[\text{versionOf}(\hat{x}, x) \wedge \text{versionOf}(\hat{y}, y) \supset \neg(\hat{y} \stackrel{R}{\Longrightarrow} \hat{x})]$$

If a variable x is independent of a variable y, we know that y cannot use x as an intermediary to affect some other variable or procedure. □

Example 3 *Interprocedural channel.* Suppose that we want two procedures to communicate through a specific variable. We say that a variable x is a *channel* from procedure P to procedure Q iff information flows from P to Q through x. This is captured by

$$\text{ChannelTo: proc} \times \text{proc} \times \text{var} \rightarrow \text{bool}$$

which is defined by

$$\begin{aligned}&\text{ChannelTo}(P, Q, x) \stackrel{\text{def}}{=} \\ &\quad (\exists \hat{x}, \hat{y}, \hat{z}\text{: vvar})(\exists R\text{: proc})[\text{versionOf}(\hat{x}, x) \wedge \text{varOf}(\hat{y}, P) \wedge \text{varOf}(\hat{z}, Q) \wedge \\ &\qquad ((\hat{y} \stackrel{R}{\Longrightarrow}_f \hat{x} \wedge \hat{x} \stackrel{R}{\Longrightarrow}_f \hat{z}) \vee (\hat{y} \stackrel{R}{\Longrightarrow}_b \hat{x} \wedge \hat{x} \stackrel{R}{\Longrightarrow}_f \hat{z}) \vee (\hat{y} \stackrel{R}{\Longrightarrow}_b \hat{x} \wedge \hat{x} \stackrel{R}{\Longrightarrow}_b \hat{z}))]\end{aligned}$$

for P and Q in *proc* and x in *var*. Since x is an interprocedural channel, we need not consider lateral flows whose purpose is to link interprocedural flows. We also rule out the possibility of a forward-backward flow, since this would make x a channel from P to itself. □

Example 4 *Interprocedural partitioning.* Assume that a procedure A is not intended to be connected to a procedure B, which we express by

$$\neg\text{ConnectedTo}(A, B)$$

The *ConnectedTo* relation says that, for any procedures P and Q, there is a transitive call from P to Q, or a transitive information flow from a variable referenced by P to one referenced by Q, or both. The predicate

$$\text{Calls: proc} \times \text{proc} \rightarrow \text{bool}$$

is defined recursively by

$$\begin{aligned}&\text{Calls}(P, Q) \stackrel{\text{def}}{=} \\ &\qquad (\exists p\text{: } plist)[\text{callByVR}(P, Q, p) \vee \\ &\qquad\qquad (\exists R\text{: proc})[\text{callByVR}(P, R, p) \wedge \text{Calls}(R, Q)]]\end{aligned}$$

where *plist* is a set of possible actual-formal pairings. The predicate

$$\text{ConnectedTo: proc} \times \text{proc} \rightarrow \text{bool}$$

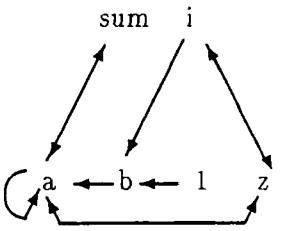
is defined by

$$\begin{aligned}&\text{ConnectedTo}(P, Q) \stackrel{\text{def}}{=} \\ &\qquad \text{Calls}(P, Q) \vee (\exists \hat{x}, \hat{y}\text{: vvar})(\exists R\text{: proc})[\text{varOf}(\hat{x}, P) \wedge \text{varOf}(\hat{y}, Q) \wedge \hat{x} \stackrel{R}{\Longrightarrow} \hat{y}]\end{aligned}$$

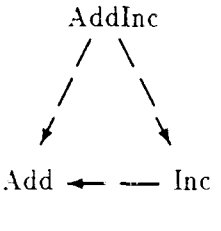
Notice that information may flow from P to Q as the result of a transitive call from P to Q (in which case R is P), or R can be a parent of P and Q that transmits a return flow from P to Q. □

```
PROCEDURE AddInc(sum,i)
ASSERT call(Add,(sum,i)) AND call(Inc,(i))
END;

PROCEDURE Add(a,b)
ASSERT affects(a,a) AND affects(b,a)
END;

PROCEDURE Inc(z)
ASSERT call(Add,(z,1))
END;
```

(a)



(b)



(c)

Figure 2: A low-level design: (a) textual representation, (b) diagram of implicit information flows, and (c) graph of call relationships.

3.2 Reasoning About Changes

It is undecidable in general to determine the exact behavioral effects of a change, but it is possible to obtain a precise, conservative approximation by formalizing the problem in terms of structural concepts. In particular, we say that a change to an object x affects an object y if the pair $\langle x, y \rangle$ is in the transitive closure of the information flow relation $\Longrightarrow$. Unfortunately, information flow is *not* transitive in the usual sense. That is, if there is flow from some object x to an object y and from y to an object z, there is not necessarily flow from x to z. As a consequence, the usual notion of transitivity gives a crude approximation of the effects of a change. To obtain a more accurate approximation of the true transitive flows (i.e., those that would occur when the system is executed), it is necessary to decompose the concept of information flow into the three special flows and to provide axioms for composing the three flows to determine the transitive closure of the information flow relation.

As an illustration of why the simple approach will not work, consider the low-level design presented in Figure 2a. The design consists of several objects: procedures *AddInc*, *Add*, and *Inc*, and variables *sum*, *i*, *a*, *b*, and *z*. Parameters are transmitted using a value-result semantics. The purpose of procedure *AddInc*, which is not specified in the figure, is to add the initial values of *i* and *sum* and return the result in *sum*; it also increments the initial value of *i* and returns the result in *i*.

We are interested, for instance, in whether or not a change to the value of variable *sum* can affect the value of variable *z*. Since two objects can interact indirectly through any number of intermediaries, the question is not whether $sum \Longrightarrow z$, but whether the pair $\langle sum, z \rangle$ is in the transitive closure of $\Longrightarrow$ on the set of all variables, written $sum \overset{*}{\Longrightarrow} z$.

We cannot form the transitive closure of $\Longrightarrow$ until the information flow relationships implicit in the *assert* statements are made explicit. The assertion for *AddInc* says that it makes two calls, one to *Add* and one to *Inc*; the ordering of the calls is unspecified for the moment. The assertion for *Add* says that the initial values of *a* and *b* affect some future value of *a*. The assertion of *Inc* specifies that it calls *Add*.

Figure 2b depicts the information flow relationships implicit in this structural description. Flows not in the figure are assumed to be invalid, since we found it useful to have a closed-world assumption [12]. For example, we assume there is no flow in *Add* from *a* to *b*, thereby making *b* a read-only variable of *Add*. Procedure calls normally cause bidirectional flow between actual and formal parameters. However, the two calls to *Add* cause only unidirectional flow from actual parameters *i* and 1 to formal parameter *b*, since the value of *b* is unchanged by *Add*.

Returning to our original question about the possibility of flow from *sum* to *z*, we can use the diagram Figure 2b to trace the information flow path

$$sum \Longrightarrow a \Longrightarrow z \tag{1}$$

from which we can infer that $sum \overset{*}{\Longrightarrow} z$. This kind of reasoning can be formalized in terms of the usual transitivity axiom.

Unfortunately, this simple analysis is much too conservative. A change to the value of *sum* cannot affect *z* if there is no execution sequence for which $sum \Longrightarrow z$. We can establish that there is no such sequence in our example specification by relating the information flow relationships in the specification to its calling relationships, which are illustrated in Figure 2c. Consider the call from *AddInc* to *Add*. Procedure *Add* contains no procedure calls and it does not allow the value of formal *a* to affect the value of its other formal *b*. Therefore, the call from *AddInc* to *Add* can only affect the value of *sum* by means of the path

$$sum \Longrightarrow a \Longrightarrow sum$$

Procedure *AddInc* also calls *Inc* with actual parameter *i*. But since *i* is never affected by *sum* (by the closed-world assumption), the call from *AddInc* to *Inc* cannot result in a flow from *sum* to *z*; from this we can conclude that the call from *Inc* to *Add* cannot as well. This completes an informal argument that $\neg(sum \overset{*}{\Longrightarrow} z)$.

To formalize this kind of reasoning, a new axiomatization of the transitivity of information flow was developed in which a transitive flow is inferred from two individual flows only when there is a *causal relationship* between the individual flows.

That is, we establish that some change in the value of variable x in the flow $x \Longrightarrow y$ causes a change in the value of variable z in the flow $y \Longrightarrow z$ before we can infer the transitive flow $x \Longrightarrow z$.

Let $\mathcal{T}$ denote a logical axiomatization of transitive information flow that takes into account the causal relationships among individual flows. In addition, let $\mathcal{S}$ denote a structural specification, and let $\mathcal{I}$ denote axioms for inferring the flows implicit in specifications. To reason about changes, we have defined $\mathcal{T}$ and $\mathcal{I}$ so that the transitive closure of $\Longrightarrow$, namely,

$$\{\langle x, y\rangle | \ \mathcal{T} \cup \mathcal{I} \cup \mathcal{S} \vdash x \Longrightarrow y\},$$

contains the true information flows in $\mathcal{S}$ for systems containing the basic structural features discussed at the beginning of this section. Moreover, the derivation of the closure should terminate for any specification $\mathcal{S}$. The transitive closure with respect to a given specification $\mathcal{S}$ serves as the basis for answering questions about changes to $\mathcal{S}$.

3.3 Controlling Connections Through Proofs

Given a structural specification $\mathcal{S}$ of a system, we would like to know that $\mathcal{S}$ accurately describes its implementation. This too is undecidable in general, but again we can obtain a good conservative approximation.

The proof strategy blends program flow analysis and formal program verification techniques. It suffices to must show that each level in a structural design hierarchy is a logical consequence of those primitive structural relations that are true of the program. Objects in a specification are connected to program objects with a mapping similar to the one in Hoare [7]. The primitive relations satisfied by a program are derived from the program automatically with a slightly modified version of standard program flow analysis techniques [1]. The flow analysis is conservative; for example, all predicates in the program are treated as uninterpreted symbols. Given the derived primitives and the mappings, the problem of consistency can be reduced to proving one or more logical implications in a typed first-order logic, where a type is a finite and fixed set.

3.4 The Initial PegaSys Prototype

PegaSys is a display-oriented, interactive environment that uses intuitive graphical pictures as formal documentation to facilitate life cycle activities for large software systems. PegaSys has been designed to offer the advantages of mathematical rigor even though users interact with it through pictures. For example, PegaSys provides standard graphical operations, via a mouse and pointing device, for manipulating

pictures, while at the same time enforcing semantic constraints on these operations sufficient to guarantee that they make sense in terms of system design. As a result, PegaSys users can document and explain system designs in a highly visual and intuitive manner.

Because of their intuitive appeal, pictures have been used frequently by computer scientists in textbooks, professional publications, and on blackboards to explain system structures. However, pictures tend to be inadequate as a means of documentation because the contain imprecise concepts that can be confusing and misleading. For instance, the same icon is often used to represent a process, a subprogram, and a data structure, all in the same picture. Similarly, the same arrow may represent the flow of data to a subprogram, the flow of control to a subprogram, or the writing of data to a data structure, all quite distinct concepts. Failure to make such distinctions might be satisfactory in a high-level design, but is not acceptable for detailed design refinements that serve as the basis for system evolution.

The goal of the PegaSys system research has been to demonstrate that it is possible to effectively support the formal specification and analysis of implementation structures. The approach has been to make use of pictures to simplify specifications and to take advantage of decidability to eliminate the need for user involvement in proofs.

The initial PegaSys prototype was extremely effective in creating the illusion that logical formulas did not exist, thereby providing the advantages of formal methods but not the drawbacks. The PegaSys prototype deals with pictures that represent *direct* connections among such objects as variables, types, procedures, and modules in sequential systems. As explained in an earlier section, we have since extended our specification technique to include indirect relations. Lower-level objects, such as statements and expressions, are not modeled. A system design is a hierarchy of such pictures, and a PegaSys user must specify the mapping between levels in a design. Given this mapping, PegaSys can prove that the design levels are consistent with each other. This prototype also supported programming in Ada and connected pictures to Ada programs.

We are presently planning a new implementation of PegaSys that incorporates the advances we have made in its underlying technology. The initial PegaSys prototype was written in in Interlisp-D on (now obsolete) Xerox 1100-series personal computers. The new implementation would be better engineered, written in portable Common Lisp on Sun workstations, and would make use of standard components whenever possible. We believe that the planned version of PegaSys would represent an important step in the introduction of formal methods in the engineering of real software systems.

4 Related Research

1. The PegaSys system is unique in its use of graphics to mask details in the formal design and proof of structural system properties. Other graphical systems have very impoverished structural design languages (see below) and do not perform a semantic analysis of designs. An interesting graphical system for behavioral specification has recently been developed by Harel [3].

2. Previous work on structural design languages falls into the following categories:

   - **Programming languages.** Block structure, import/export lists, and encapsulation mechanisms have been used to specify referencing environments in several programming languages, including Euclid, Mesa, Modula-2, Ada, and PIC/Ada. These features describe access to objects but not the use of objects. Furthermore, they deal solely with program-level objects. Module interconnection languages have essentially the same drawbacks.
   - **Program design languages.** Over a dozen structural design languages have been developed since they were made popular by Yourdon, De Marco, and others; see [9] for a survey. These languages typically represent program structure as a directed graph in which nodes denote program objects and arcs denote structural relations among objects. Relations in a graph are low level and not formally defined. Moreover, there is no mechanism for properly defining new relations; thus, it is not possible to formalize many common design abstractions.
   - **Formal specification languages.** These languages, e.g, Anna [8], Larch [6], and OBJ [5], focus on behavior, not structure. Some contain a form of import/export list.
   - **Derivational techniques.** Program transformations describe how to transform a given structure into a different and possibly more efficient structure. A related technique presently gaining much attention involves a use of constructive type theory in which programs are correct by construction. Both approaches have been applied primarily to algorithm design. In contrast, our approach is focused on how algorithms are put together to form a system.

3. Previous work on the analysis of system changes is either at too low a level or limited by undecidability:

   - **Program inspection.** In practice, the dominant way of determining the effects of a change is for the human to interpret various relations extracted

from the program itself. The relations include direct (cross-reference) relations and transitive relations about calling relationships and data flows. Numerous tools have been developed which derive such relations, but they are used primarily for other purposes, such as program optimization [2], the detection of simple errors [4], and documentation [10]. We are not aware of any existing tool that provides a systematic way of combining the relations to determine the effects of changes.

- **Semantic proofs.** In the context of formal program verification, Moriconi [11] developed a general approach to reasoning about the semantic effects of changes. Given a functional specification of a system and a Hoare-style logic, the method determines what formulas must be proved to isolate the *exact* semantic effects of incremental changes. Unfortunately, these formulas are in an undecidable theory, and experience indicates that they cannot be proved without substantial human assistance. Consequently, the approach is impractical for everyday use.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] K.D. Cooper, K. Kennedy, and L. Torczon. The impact of interprocedural analysis and optimization in the $\mathbf{R}^n$ programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):491–523, October 1986.

[3] D. Harel *et al.* STATEMATE: A working environment for the development of complex reactive systems. In *Proceedings of the 10th International Conference On Software Engineering*, pages 396–406, Singapore, April 1988.

[4] L.D. Fosdick and L.J. Osterweil. Data flow analysis in software reliability. *Computing Surveys*, 8(3):305–330, September 1976.

[5] K. Futatsugi, J. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *Proceedings of the Twelfth Symposium on Principles of Programming Languages*, pages 52–66, Association for Computing Machinery, 1985.

[6] J.V. Guttag, J.J. Horning, and J.M. Wing. The Larch family of specification languages. *IEEE Software*, 2(5):24–36, September 1985.

[7] C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[8] D.C. Luckham and F.W. von Henke. An overview of Anna, a specification language for Ada. *IEEE Software*, 2(2):9–23, March 1985.

[9] J. Martin and McClure C. *Diagramming Techniques for Analysts and Programmers.* Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[10] L.M. Masinter. *Global program anulysis in an interactive environment.* Technical Report SSL-80-1, Xerox Palo Alto Research Center, Palo Alto, California 94304, January 1980.

[11] M. Moriconi. A designer/verifier's assistant. *IEEE Transactions on Software Engineering*, SE-5(4):387–401, July 1979. Reprinted in *Artificial Intelligence and Software Engineering*, edited by C. Rich and R. Waters, Morgan Kaufmann Publishers, Inc., 1986. Also reprinted in *Tutorial on Software Maintenance*, edited by G. Parikh and N. Zvegintzov, IEEE Computer Society Press, 1983.

[12] R. Reiter. On closed world data bases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 55–76, Plenum Press, New York, New York., 1978.

[13] C.E. Shannon. A mathematical theory of communication. *Bell Systems Technical Journal*, 27:379–423 (July), 623–656 (October), 1948.

Visualizing Program Designs Through PegaSys

Mark Moriconi and Dwight F. Hare, SRI International

PegaSys is concerned more with explaining program design than describing programs, and offers more extensive support to programming in the large than other graphical systems.

This article is an introduction to many of the interesting features of PegaSys,* an experimental system that encourages and facilitates extensive use of graphical images as formal, machine-processable documentation. Unlike most other systems that use graphics to describe programs, the main purpose of PegaSys is to facilitate the explanation of program designs.

A program design is described in PegaSys by a hierarchy of interrelated pictures. Each picture describes data and control dependencies among such entities as "subprograms," "processes," "modules," and "data objects," among others. The dependencies include those represented in flowcharts, structure charts, dataflow diagrams, and module interconnection languages. Moreover, new abstractions can be defined as needed.

What is particularly interesting about PegaSys is its ability to: (1) check whether pictures are syntactically meaningful, (2) enforce design rules throughout the hierarchical decomposition of a design, and (3) determine whether a program meets its pictorial documentation. Much of the power of PegaSys stems from its ability to represent and reason about different kinds of pictures within a single logical framework. This framework is transparent to PegaSys users in the sense that interactions are in terms of icons in pictures. For example, formal properties of a program are described by standard graphical operations on icons rather than by sentences written in a formal logic.

Excerpts from a working session with PegaSys are used to illustrate the basic style of interaction as well as the three PegaSys capabilities.† We describe the key ideas behind PegaSys elsewhere.[2,3]

Background and related work

Pictures have been used extensively by computer scientists in textbooks, professional publications, and on blackboards to explain dependencies in programs. Although pictures may be quite perspicuous, they have tended to be inadequate as a means of documentation. One reason is the use of imprecise concepts that result in pictures that are confusing and easily misinterpreted. For example, the same graphic symbol is often used to represent a process, a subprogram, and a data structure, all in the same picture. Similarly, an undifferentiated arrow might represent the flow of data to a process, the flow of control between subprograms, or the writing of data

* Programming Environment for the Graphical Analysis of SYStems.

†This article is a condensed version of a paper contained in a technical report.[1] Because of space limitations, we have removed many of the pictures that describe the design of the example system developed during the session.

0018-9162/85/0800-0072$01.00

into a data structure, all quite distinct concepts.

While formal documentation does not suffer from this imprecision, its advantages have tended to be outweighed by the difficulty of constructing and understanding it. Moreover, formal documentation has inadequately captured dependencies among components of the program it is intended to describe. An understanding of such dependencies is crucial throughout the software life cycle, especially during maintenance, and becomes increasingly more difficult to glean from a program as it increases in size and complexity.

In light of these observations, PegaSys attempts to take advantage of pictorial communication in describing data and control dependencies while, at the same time, maintaining the advantages of mathematical rigor. PegaSys is differentiated from previous graphical systems by its wider range of representation and analysis and its more extensive support for programming in the large. Previous work most closely related to PegaSys is concerned with representation and analysis techniques. We review this work and then describe related systems.

For a system to perform any sort of meaningful analysis of a picture, it must maintain a logical representation of the picture. A number of formalisms have been developed that have, or easily could have, a pictorial rendering. Examples are flowcharts,[4] structure charts,[5] pictographs,[6] dataflow diagrams (surveyed in Davis and Keller[7]), plans,[8] and module interconnection languages.[9-12] All of these formalisms capture data and control dependencies, typically down to executable program fragments. Pictures in PegaSys describe what we believe to be the important design concepts in these formalisms, plus other concepts as well.

The presence of a logical representation for a picture provides a basis for reasoning about the picture. In addition to checks for syntax errors, two other sorts of syntactic analysis of pictures have been performed by previous systems. The first involves the hierarchical refinement of a picture. If we think of a picture as a graphlike diagram, a node in a diagram may be replaced by a diagram provided that the replacement preserves the connectivity of the original diagram. Example uses of this idea can be found in Davis and Keller[7] and Rich and Shrobe.[8] The second sort of analysis concerns the relationship between a picture and the program it is intended to describe. If a picture is not executable, it is important to verify whether it accurately describes the program. For example, the flow of control in a program can be determined purely syntactically if we assume that conditional control paths may always be executed. Similarly, the "uses" and "requires" relations in module interconnection languages can be verified using type-checking techniques.[12] In contrast, PegaSys additionally places semantic constraints on design refinements and programs.

Pictures in PegaSys describe how algorithms and data structures fit together to form the design of a larger program.

One such constraint deals with the logical consistency between a picture and the program it is intended to describe. Traditionally, program verification efforts have employed general methods for establishing the logical consistency between a formal specification and a program.[13] The PegaSys verification procedure is more specialized and simpler, and does not have the practical drawbacks of traditional approaches.

A related system that deals with program dependencies is the PECAN system.[14,15] PECAN provides multiple "views" of a program by extracting dependencies directly from a program and then displaying them graphically. A similar, albeit nongraphical, approach at the level of specifications is described in Swartout.[16] The approach taken by PegaSys differs in that the program designer is responsible for describing a program in terms of the abstractions used in its conceptualization. This approach is based on our belief that it is difficult, if not impossible, to generate these abstractions from the final program.

Other related systems that make extensive use of graphics to describe aspects of programs fall into two major categories. First, there are a number of systems for "animating" dynamic program execution, a good example of which is the Balsa system.[17] Balsa creates simulations in which sophisticated graphical representations of an algorithm and its data structures are continually updated throughout the execution of the algorithm. There are other examples of animation systems.[17-22] The second category is concerned with "visual programming," i.e., programming by spatial arrangement of icons.[23-27] Both kinds of systems have tended to focus almost exclusively on programming in the small—that is, on individual algorithms and data structures. Pictures in PegaSys, on the other hand, describe how algorithms and data structures fit together to form the design of a larger program.

Getting started

Figure 1 shows a bitmap display connected to a Xerox personal computer.* Screen real estate is divided

*PegaSys is implemented in Interlisp-D and runs on Xerox 1100-series personal computers.

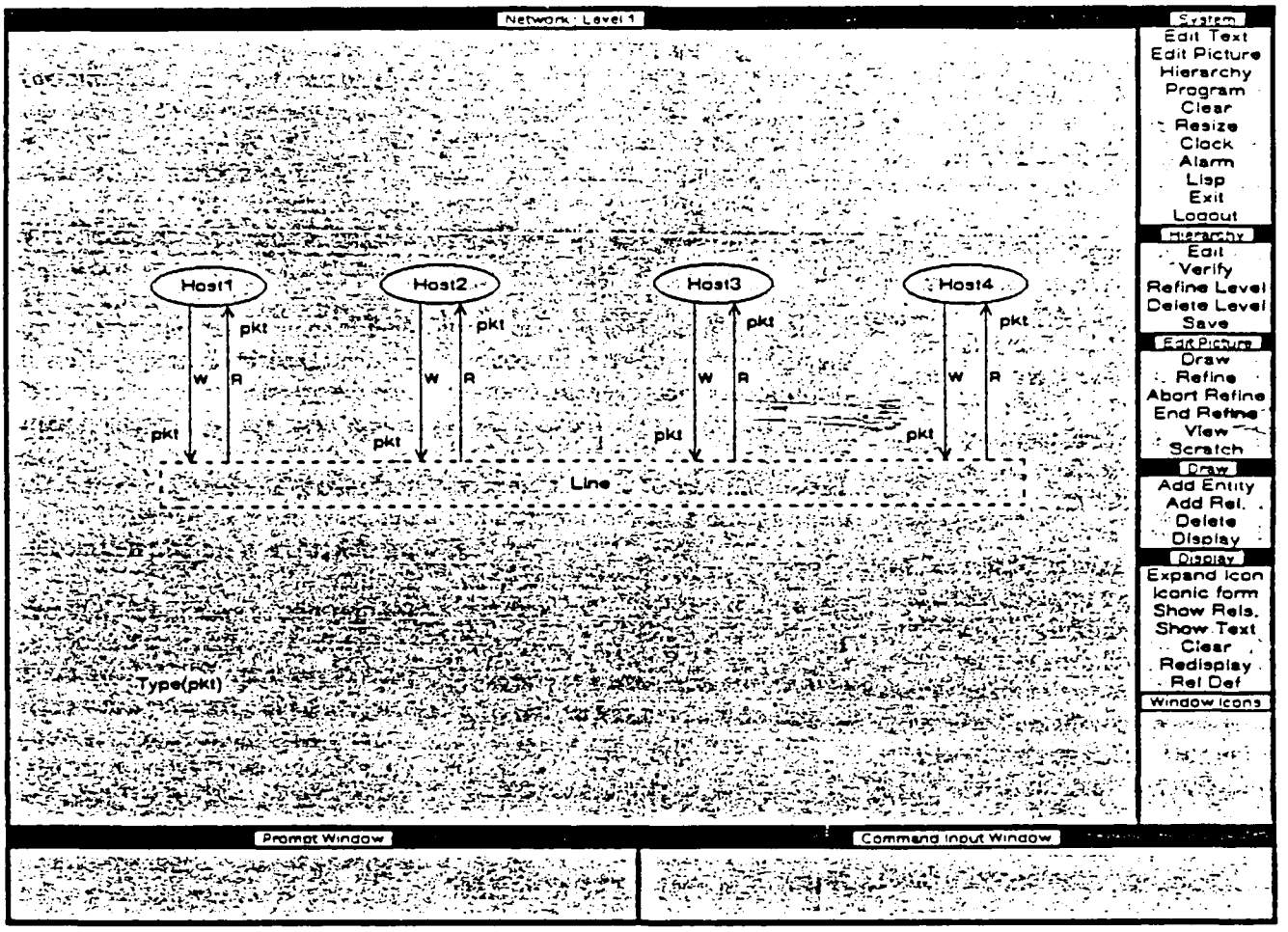


Figure 1. First level in broadcast network documentation hierarchy.

into adjacent, nonoverlapping rectangular areas called *windows*.* Screen layout will be adjusted throughout the scenario in an attempt to make maximal use of screen real estate. This is done by pointing with the mouse. The small windows down the right-hand side are *menus* containing commands, each of which may be selected by pointing at it with the mouse. The black strip at the top of each window contains the window's name. The name of a window is intended to be suggestive of its contents. For example, the name "Network:Level1" indicates that the contents of the associated window is the first level of the picture hierarchy for a network.

For the most part, arguments to commands are selected or constructed by pointing, and pictures are manipulated by pointing as well. On-line help and feedback on errors appear in the prompt window in the lower-left corner of Figure 1.

The example session used to illustrate aspects of PegaSys is concerned with the development of a realistic broadcast network. It is not necessary to understand the details of the network or its implementation in order to get a "feel" for the capabilities being demonstrated. Particularly germane aspects of the example network are explained as needed. As the session progresses, details of the network are omitted so as to focus attention on the aspect of PegaSys being described.

The session begins with the design of the overall broadcast network down to the host level. It then focuses on the development of a single host, whose multilevel design and implementation is reused several times in the overall network. The network was implemented in the Ada programming language[29] (using PegaSys) and subsequently run on a Data General MV 10000 computer.

The meaning of a picture

A crucial aspect of the PegaSys design is its treatment of a picture as

*Our display management strategy is patterned directly after the *tiling* strategy used in Cedar.[28]

both a graphical and a logical structure. These structures affect user interaction with PegaSys in several important ways.

Dual interpretation of pictures. A picture is represented as a graphical structure composed of icons and their properties, such as size and location. Icons in a picture correspond to predicates in the underlying logical representation of the picture. This logical structure captures the computational meaning of a picture; each predicate in this structure denotes a computational concept expressed by the picture.

The picture shown in Figure 1 contains several icons: four ellipses, a rectangle, several arrows, and several character strings.* These icons denote several concepts about the example network. Each of the four hosts in the network is modeled as a process (indicated by an ellipse); the communication line by a module (indicated by a dashed rectangle); and a packet of data by a type (indicated by a label on arcs).† Interrelationships among hosts, packets, and the line are described by the "write" relation (denoted by the letter W on arrows) and the "read" relation (denoted by R).

At a first approximation, the picture says that the broadcast network consists of four hosts that communicate by means of a line. More precisely, processes named Host1,...,Host4 write values of type pkt into a module called Line and read values of the same type from the Line module.

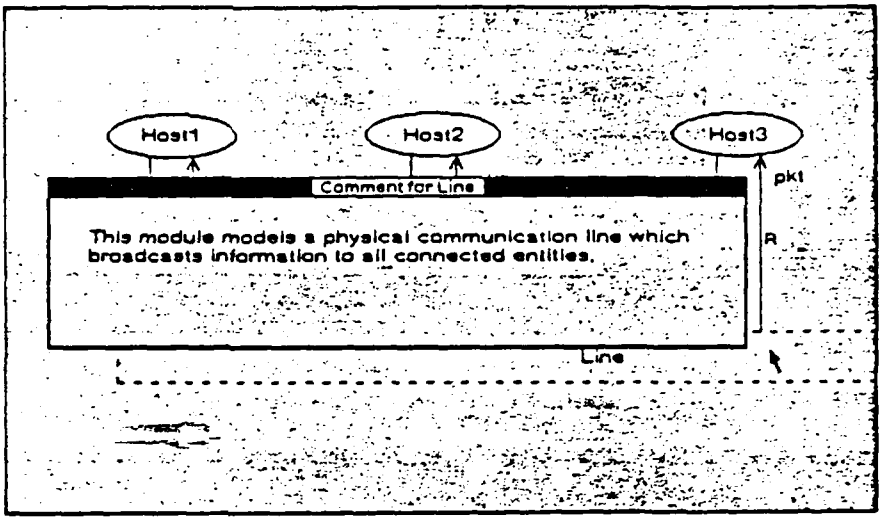


Figure 2. Explanatory text may be associated with computationally meaningful icons.

This statement about the network is represented in PegaSys by a conjunction of the predicates

process (*Host1*), *process* (*Host2*),
process (*Host3*), *process* (*Host4*),
module (*Line*), *Type* (*pkt*),
write (*Host1*, *Line*, *pkt*),
Read (*Host1*, *Line*, *pkt*)

with similar *Write* and *Read* predicates involving *Host2*, *Host3*, and *Host4*. Notice that every predicate corresponds to a different icon in Figure 1. Purely cosmetic changes to a picture, such as an adjustment to the size or location of an icon, do not require updates to the logical representation of the picture.

The logic in which pictures are represented is called the *form calculus*. A syntactically correct picture is said to describe the *form of a program* and is represented by a well-formed formula of the form calculus.

The following terminology will be adopted to refer to components of a picture. *Active entities* may initiate actions that create, destroy, or transform data objects (variables); the data objects themselves are called *passive entities*. The existence of an active or passive entity is determined by its membership in a defining relation. An example of an active entity is *process* (*Host1*), and an example of a passive entity is *Type* (*pkt*). The term *entity* refers to both kinds of entities. A relationship among entities, such as specified by the *Write* relation, is called an *interaction*.

Entities and interactions specified in pictures correspond to either primitives of the form calculus or predicates defined in terms of the primitives. The primitives were carefully chosen to facilitate the definition of new concepts.[2]

A brief summary of the primitives will suggest the general kinds of concepts that can be expressed by pictures in PegaSys. Active entities are specified by "subprogram," "process," and "module" relations. We have chosen this relatively course grain in an attempt to capture the salient aspects of the design of a program, as opposed to the details of particular algorithms. Passive entities are specified by a "name" relation and by "simple type" and "structured type" relations. A name is used to refer to the object and a type to denote a (possibly structured) value set. The manipulation

*Note that type pkt is represented by text in the lower left of the picture rather than by an icon. If PegaSys does not have an appropriate icon for a concept, the convention is to display its logical representation as text.

†A *process* sequentially executes a series of actions that may proceed in parallel with actions of other processes; a *module* is a collection of one or more logically related entities; and a *type* is a, possibly structured, value set. The line is not modeled as a process because its actions are initiated by hosts

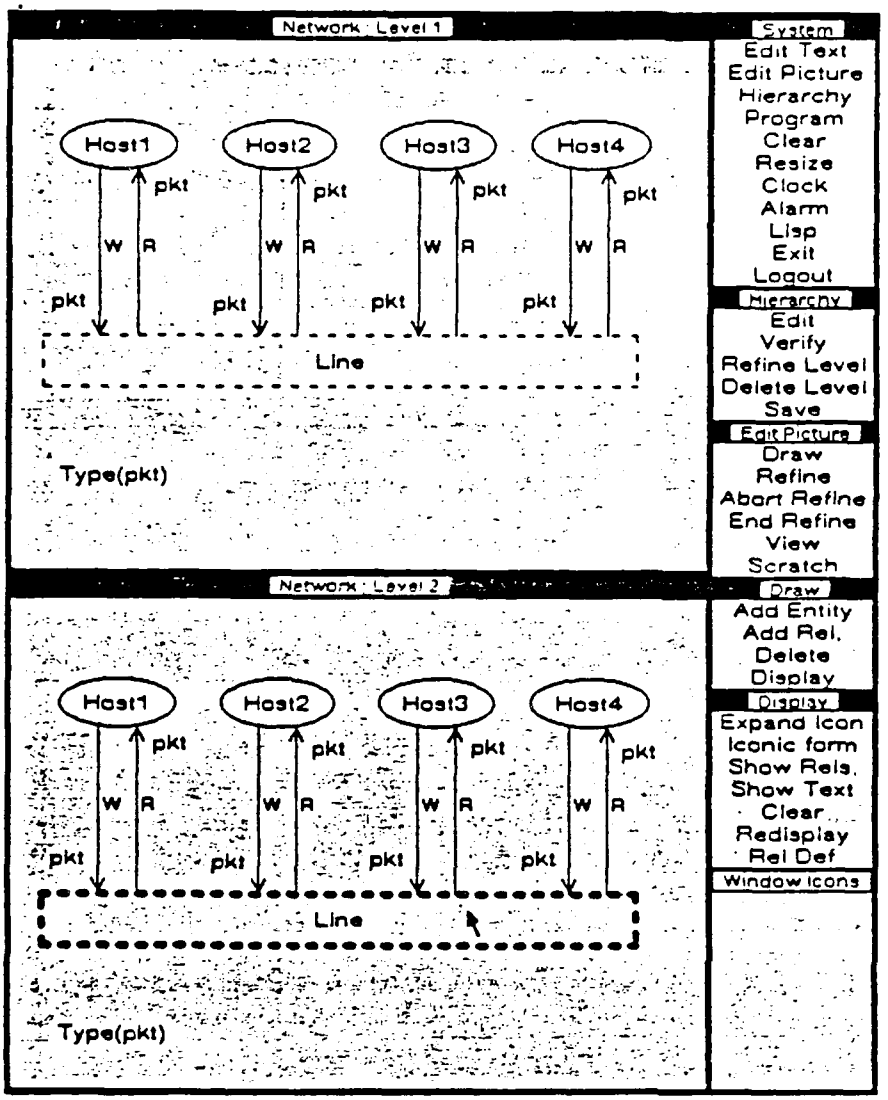


Figure 3. Creation of a level and selection of the line for refinement.

and sharing of data objects are specified by means of primitive interaction relations that capture general notions of data object declaration, data object visibility, aliasing of names, modification of the value of a data object, and accessing the value of a data object. There are also primitives for modeling (synchronous and asynchronous) interprocess communication and ordinary transfer of control. See reference 3 for details.

Finding out about what is not in a picture. A picture may be augmented with explanatory text. In particular, text may be associated with any computationally meaningful icon. If the user points at an icon and presses a button on the mouse, the associated pop-up comment will appear on the display until the user releases the button. Figure 2 shows the *pop-up comment* for the line module. Given this and related features of PegaSys, the best way to gain an understanding of the pictures presented here is by means of an interactive dialog with PegaSys.

Manipulation of pictures

Interactions with PegaSys are in terms of icons. However, graphical operations on pictures are restricted by logical constraints imposed by the form calculus. These constraints are intended to ensure that graphical operations make sense computationally.

Graphical manipulation of pictures. Graphical manipulation of a picture depends upon a one-to-one mapping between computationally meaningful icons and predicates. An icon and its associated predicate denote the same concept.

Perhaps the simplest example of how PegaSys takes advantage of this mapping concerns the selection of a concept, which is done by pointing at the appropriate icon. For example, positioning the mouse to point at the ellipse labeled Host1 in Figure 1 and clicking (depressing and releasing) a button on the top of the mouse results in selection of the predicate *process* (*Host1*).

Another example concerns the construction of pictures. Pictures are constructed by using a series of graphical operations on the display that have the twofold effect of building a graphical structure and a corresponding formula in the form calculus. Each operation involves the selection of a concept from a menu followed by its placement at a location on the screen. An icon is associated automatically with most concepts. If a concept must be named, the user must enter a name for it and PegaSys will size the associated icon relative to the size of the name. Placement is done by pointing. Layout adjustments may be made by pointing at the desired icon (selection), pointing at the destination location, and clicking a button on the mouse. PegaSys repositions the selected icon at the specified location, readjusting related icons (such as connected arrows) as best it can.

Logical constraints on graphical manipulations. Both syntactic and semantic constraints are placed on graphical manipulations. An example of the former concerns the construction of pictures. While pictures are constructed by means of standard graphical operations, the form calculus guides the entire process. PegaSys

uses the *grammar* of the form calculus to guide the construction of pictures in much the same way that a structure-oriented editor uses the grammar of a programming language to guide the construction of programs. Pictures may contain only concepts that are primitive in the form calculus or that have been defined in terms of the primitives. PegaSys uses the *type constraints* on predicates to prevent a nonsensical composition of concepts. For example, if a predicate has been defined to take two processes as its arguments, PegaSys ensures that both arguments are provided and, moreover, that both are processes. If not, a type error is signaled.

Semantic constraints are needed to restrict picture refinements and to analyze the relationship between a picture hierarchy and the program it is intended to describe. In both instances, it is necessary to prove logical formulas in the form calculus. However, this can be done quickly and without user interaction (due in part to the decidability of the form calculus).

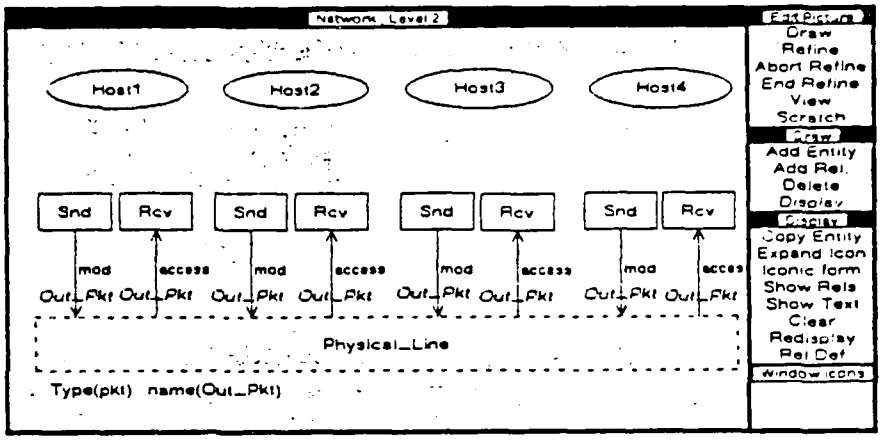


Figure 4. Constructing a replacement for the line.

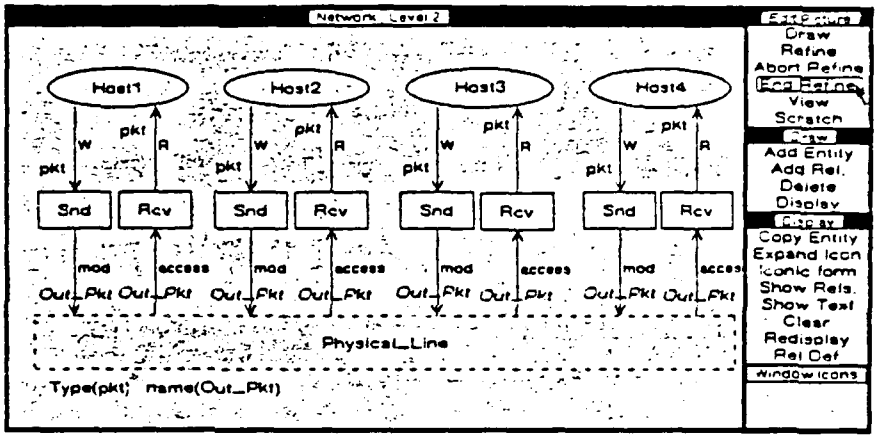


Figure 5. Picture at level 2 after refinement of the line.

Hierarchical decomposition of pictures

A hierarchy of pictures related according to the PegaSys design rules is said to describe the design of a program.

Creating a new level in a hierarchy. Each level in a picture hierarchy is a description of a program at a particular level of detail. A level is formed by a sequence of refinements to the immediately preceding level in the hierarchy. A refinement adds detail to an existing concept and is not allowed to delete concepts from a picture. Therefore, a concept cannot appear at any level in a hierarchy (except the top one) unless it is a refinement of a more abstract concept.

The procedure for building a new level in a hierarchy is as follows. As soon as the user indicates a desire to create a new level, PegaSys makes a copy of the immediately preceding level. The new level is formed by a sequence of refinements to this copy. An individual refinement involves the following three steps: (1) selection (by pointing) of the relation to be refined, (2) construction or selection of its replacement, and (3) selection of the appropriate menu command.* PegaSys checks whether the refinement satisfies its design rules.

Refinement of an active entity. Recall that an active entity is an entity that has the ability to manipulate data. The active entities in Figure 1 are the host processes and the line module. The next step in the scenario illustrates a refinement technique called *active entity refinement*—the first, and simplest, of three refinement techniques employed in the network development.

Provided the replacement preserves interactions involving the replaced entity, PegaSys allows an active entity to be replaced by a picture. The three steps in an active entity refinement are illustrated by Figures 3 through 5. The window at the bottom of the display (see Figure 3) contains a copy of level 1, where the Line module has been se-

*This is a good example of the modeless style of interaction supported by PegaSys in that argument selection precedes command selection. See Tesler's discussion of this approach to man-machine interfacing.[30]

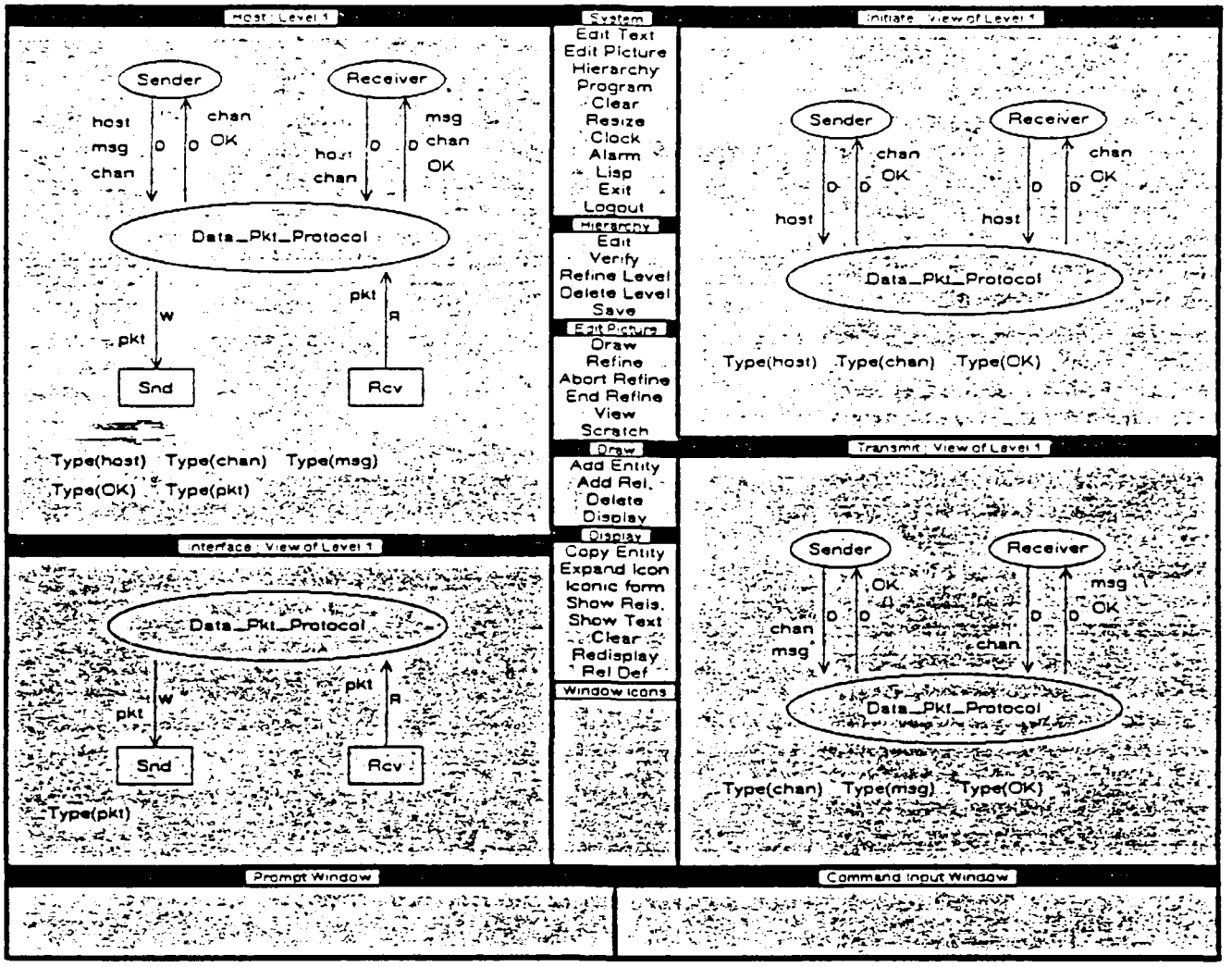


Figure 6. Level 1 of protocol hierarchy for a host computer. The upper-left window contains a picture of the entire level, which is explained by the views in the other three windows. This figure is read starting with (a) for the upper-left window and progressing clockwise for windows (b) through (d).

lected for refinement (indicated by boldface highlighting). The user next constructs a replacement for the line, as shown in Figure 4. Solid-lined rectangles denote "subprograms" that are intended to specify the interface to the line; Out_Pkt is a variable modified by these subprograms.* Finally, the user indicates (by pointing) its exact connection between the replacement for the line and the hosts (see Figure 5). This completes the refinement, and PegaSys checks that interactions at level 1 are preserved at this level and that the entire picture at level 2 satisfies the type constraints imposed by the form calculus. (PegaSys accepts only well-formed pictures and, therefore, requires that type errors be removed before it attempts any further analysis of the offending picture.)

Figure 5 also illustrates a useful aspect of picture layout in PegaSys. Even though the Snd and Rcv entities appear four times each at level 2, they describe only one interface to the line and, therefore, appear only once in the internal logical representation of the picture. In general, duplication is a good technique for avoiding crossover and curved lines.

Views are used to manage complexity. We are now ready to design the hosts in the network. Rather than designing four separate hosts, our strategy will be to design *one* host and then "replicate" it four times at level 2 of the network hierarchy (see Figure 5).

Figure 6a shows the first level in the design hierarchy for a host. This picture is not particularly perspicuous because it mixes several important properties of a host. These properties may be separated by means of three

*The access and mod relations in Figure 4 require Out_Pkt to be a variable belonging to the physical line module.

views (explained below), obviating the need to study Figure 6a.

In general, multiple views of the same picture are used to manage complexity or to emphasize particular aspects of a picture. A *view* in PegaSys is a single grouping of logically related icons from a picture. Views are presently constructed interactively by structured selection and positioning of related icons. A more sophisticated view mechanism, based on relational database technology, is planned.

Two of the views describe the two steps in interhost communication—namely, establishment of a communication link (i.e., a channel) between hosts (Figure 6b) and transmission of an actual message (Figure 6c). In Figure 6b, a sender process asks a data packet protocol to open a channel between it and another host. (A single host may have multiple open channels.) If the channel is successfully opened, the variable OK has the value true and chan contains the name of the open channel. If the attempt to initiate a connection fails, OK has the value false. The receiver opens a channel in the same manner.

Figure 6c describes message transmission. The sender sends a message (msg) over the open channel (chan) and receives back an indication as to whether the transmission was successful. The receiver, on the other hand, tells the data packet protocol that the channel called *chan* is open and awaits the arrival of a message.

The third view, shown in Figure 6d, describes the interface between a host and an external network backbone. This view says that a host reads and writes packets by means of subprograms Rcv and Snd, respectively. This interface would be suitable for a variety of network configurations, including the line interface in Figure 5. We will say more about this interface later when we "paste" the completed host design into our network.

Refinement of an interaction. We have seen one example of how refinements add detail to an existing design concept. In particular, a refinement of an active entity adds detail in the sense that it syntactically elaborates the entity and preserves interactions in the design. (Pictures only suggest the semantics of entities by means of mnemonic entity names. It is expected, but in no way enforced, that the refinement of an entity provide a more detailed description of the computation suggested by the entity name.) For refinements of interactions, it is possible to enforce stringent logical requirements—in particular, a refinement of an interaction must be a more detailed description of the interaction. For example, if a picture says that an entity "writes" into a particular data object, then refinements of the notion of writing must specify one of the possible ways in which writing may occur.

The sequence of steps performed in refining an interaction are illustrated in Figures 7 through 10. The user first selects a dataflow relation D (see Figure 7). Its replacement is constructed by selecting the menu command for adding a relation and then the Write relation from a pop-up menu (see Figure 8). Note that the dataflow relation has disappeared while it is being refined. The Write relation takes three arguments, two of which are selected in Figure 8. The two selections

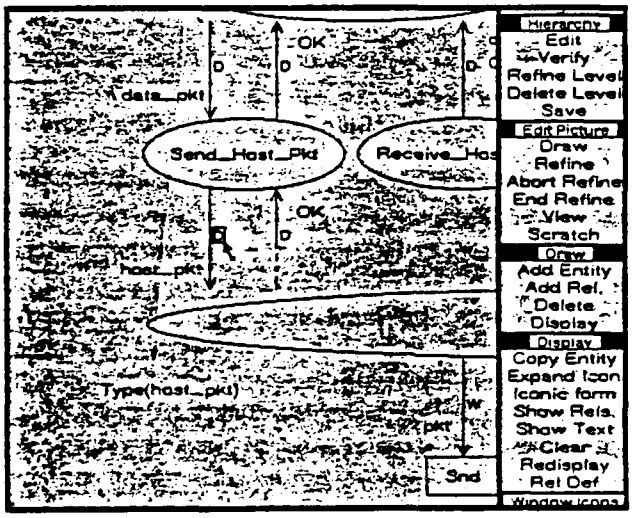


Figure 7. Selection of an interaction relation for refinement. The letter *D* is an abbreviation for a dataflow relation.

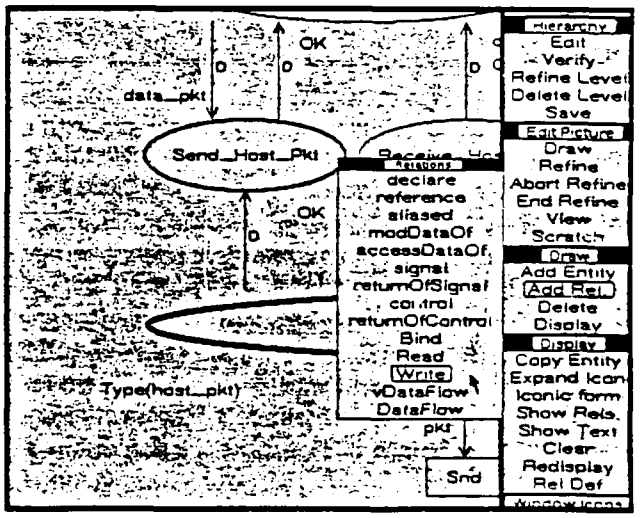


Figure 8. Selection of the Write relation from a pop-up menu to replace the dataflow relation selected in Figure 7.

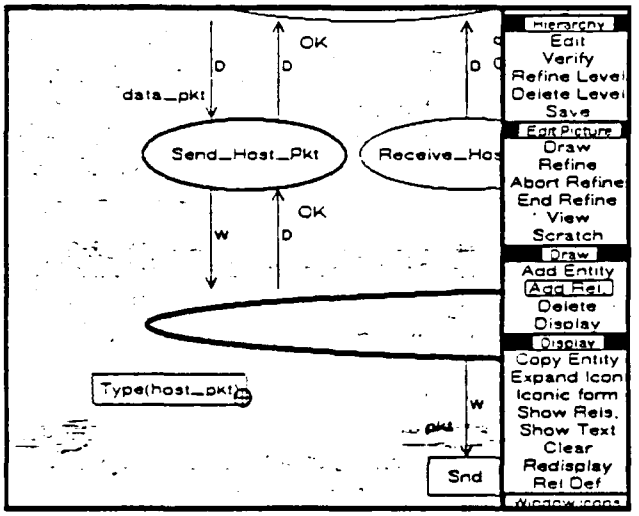


Figure 9. Selection of an argument to the Write relation. The cursor has changed to prompt for a selection.

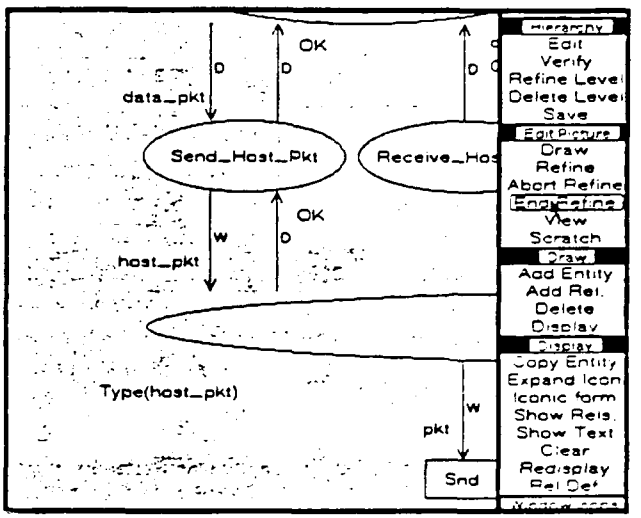


Figure 10. The dataflow relation of Figure 7 has been replaced by the Write relation (abbreviated as W). Validation of this refinement required a logical proof.

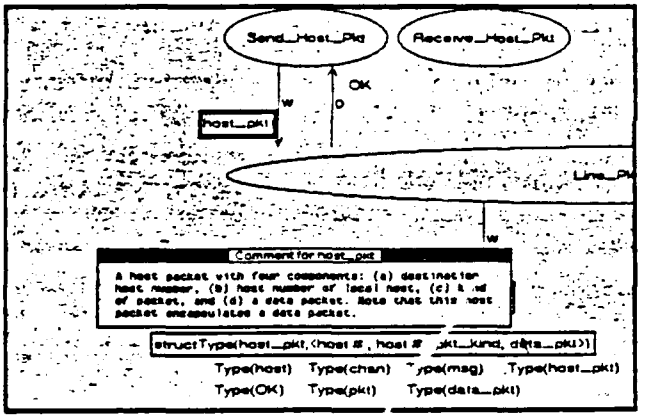


Figure 11. Selection of passive entity host_pkt with a pop-up comment explaining its selected refinement.

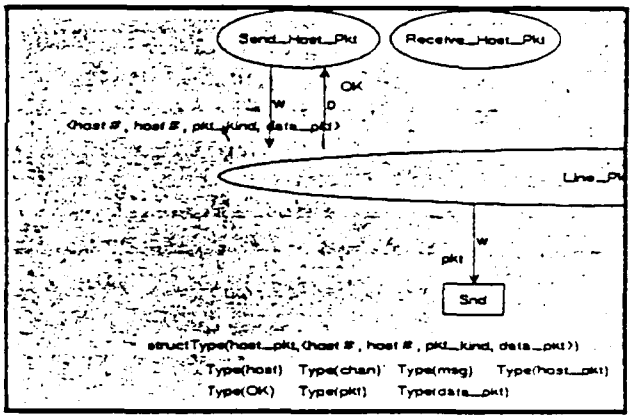


Figure 12. The result of the refinement started in Figure 11.

are the bold ellipse (whose name Line_Pkt_Protocol is occluded by menus) and Send_Host_Pkt. The third argument is selected in Figure 9. (The cursor has changed to let the user know that a selection is required.) The final result is seen in Figure 10, which specifies that a data object of type host_pkt is written from Send_Host_Pkt to Line_Pkt_Protocol.

PegaSys allowed the replacement of the DataFlow relation by the Write relation because it was able to prove a certain logical relationship between them. Roughly speaking, the refinement of an interaction is said to add detail if the interaction is a logical consequent of its refinement. This logical relationship is verified by means of a (fully automatic) logical proof.*

*This procedure applies to any derived relation, while the active entity refinement strategy applies only to primitive active entities.

Refinement of a passive entity. Recall that a passive entity is a data object manipulated by active entities. A data object is characterized by a name (which is used to refer to the object) and a simple or structured type (which denotes a set of possible values). A passive entity refinement, unlike refinements described earlier, does not necessarily replace an existing relation; it usually augments a *partial characterization* of a data object. The simplest example is the addition of a miss-

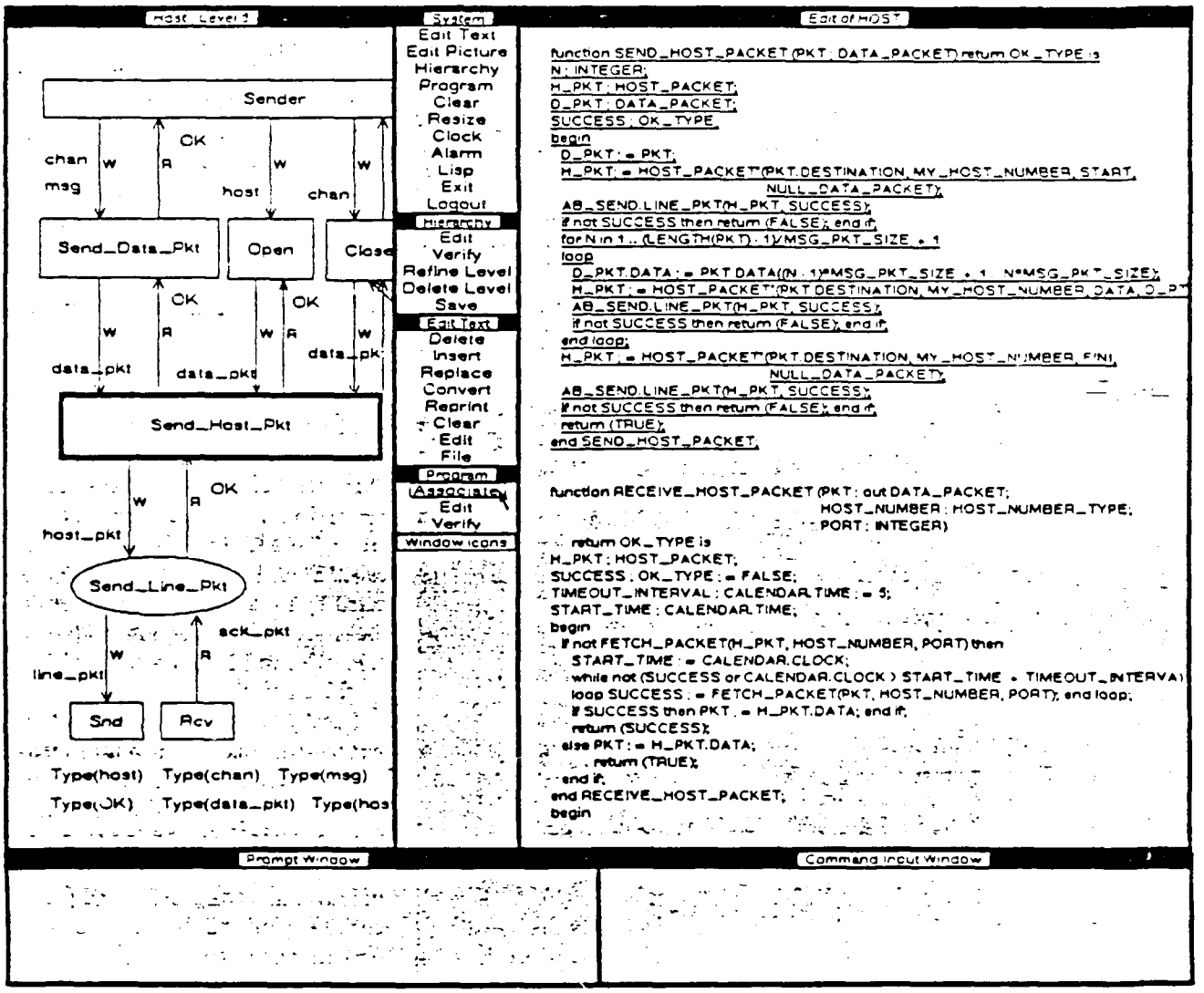


Figure 13. Associating entities in pictures with program text.

ing name or type to a data object. A more complex example involves the specification of the structure of a global type. (All types are global.) It is often convenient to specify different *instances* of the *same* type differently. In particular, only the relevant components of a structured data object need be specified for each instance of the object.

The refinement of a single instance of a type is illustrated in Figures 11 and 12. In Figure 11, the user has selected the type host_pkt (indicated by the bold rectangle), constructed the refined structure (the structType relation at the present position of the cursor), and entered the pop-up comment explaining the relation. Figure 12 shows the completed refinement. This refinement of host_pkt into a four-tuple applies only to the selected instance of type host_pkt. Components of the host_pkt structured type, such as host# and pkt_kind, can be further refined into structures and substructures using the structType relation.*

Sometimes it is convenient to refine an instance of a simple type into another simple type, rather than a struc-

*In order to avoid clutter on the display, simple types are not actually replaced by structured types in pictures. For example, *host_pkt* was not actually replaced in the picture by the four-tuple describing its structure. However, if the user points at host_pkt (on the arc between Send_Host_Pkt and the partially occluded ellipse) and presses a button on the mouse, the specified structure is displayed (see Figure 12).

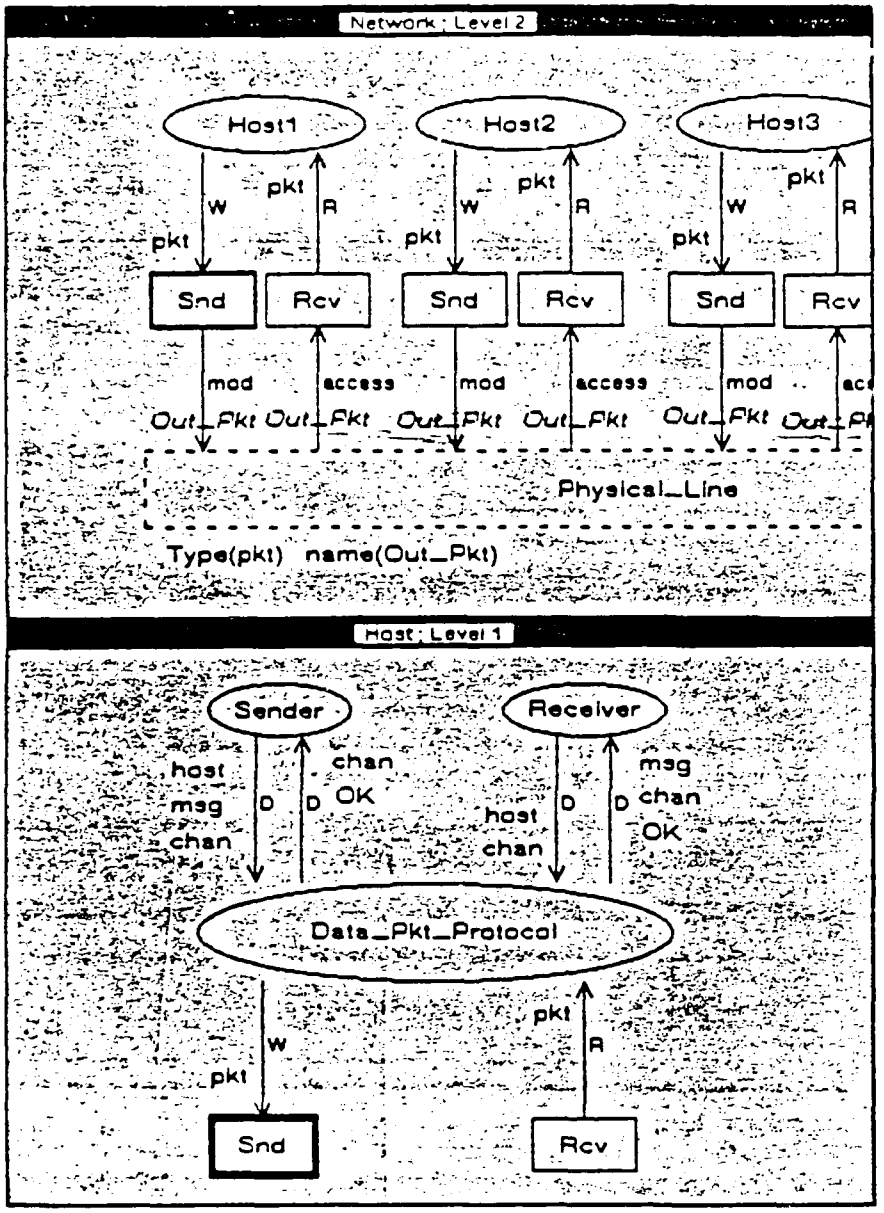


Figure 14. A design is shared through its interface entities, which are connected to another design by pointing.

tured type. This is done if a single type is used to denote a *union of types*. For example, we use the type pkt in our network as an abstraction for two different kinds of packets, a line packet and an acknowledgment packet. PegaSys allows us to replace pkt by line_pkt in one refinement and by ack_pkt in another. (Eventually, line packet line_pkt is refined into < host#, host#, seq, host_pkt >. This packet encapsulates host packets and contains an acknowledgment bit (of type seq) required by the line-level protocol. An acknowledgment packet is refined into < host#, host#, seq >.)

Reasoning about programs

We omit the development of the remaining levels in the host hierarchy (which are described in reference 1) and rejoin the session. We are now ready to implement the host in Ada and to verify that its implementation is logically consistent with the host design. Programs are written interactively using the PegaSys structure-oriented editor; the verification process does not require human intervention except to establish the correspondence between entities in a picture and program constructs.

Each correspondence is specified by two structured selections, one from a picture and one from a program. This is illustrated in Figure 13, where the user has selected an entity called Send_Host_Pkt (indicated by the bold rectangle) and an Ada program unit called SEND_HOST_PACKET (indicated by the underlined text). Issuing the Associate menu command (see the cursor) causes PegaSys to record the specified association.

PegaSys requires that each *atomic entity*—i.e., one that is not refined—must be associated with exactly one program construct. Active entities must be associated with program units (in the case of Ada, a subprogram, package, task, or generic) and passive entities with data object or type declarations. The kind of an entity determines what it can be associated with. This association may occur at any stage of the development and at any level in a design hierarchy.

Nonatomic entities are not allowed to be associated with program constructs. We just saw that the type abstraction called pkt was replaced by line_pkt and ack_pkt. It does not make sense to require type pkt to be represented in the program, only that line_pkt and ack_pkt be represented. However, in general, there are situations in which it would be desirable to associate nonatomic entities with program constructs. The association would have to be restricted based upon properties of the refinement history. PegaSys maintains this history but

does not as yet provide this capability.

Once a program construct has been associated with every atomic entity in a hierarchy, PegaSys can attempt to prove that the program and the hierarchy are logically consistent. That is, PegaSys proves that the *lowest level* in a hierarchy is logically consistent with the program it is intended to describe. (This does not mean that an entire hierarchy is consistent with a program. PegaSys shows that every level in a hierarchy follows from the immediately preceding level by valid applications of our refinement rules and that the lowest level is consistent with the program it is intended to describe.) The PegaSys proof procedure has the following two important characteristics: (1) properties of nested program units are inherited by their parents and (2) specified interactions can be satisfied in a variety of ways by an implementation.[3] Without these considerations, impractical constraints would be placed on an implementation.

It should be pointed out that PegaSys is actually proving that a picture is logically consistent with a program under a *reasonable interpretation* of the program. PegaSys presently assumes that the consistency between a picture and the program it is intended to describe does not depend upon certain properties of its implementation. For example, it assumes that consistency does not depend upon "dead" control paths or aliasing of names in the same context. For the sorts of properties described by pictures in PegaSys, the assumptions appear to be reasonable and to coincide directly with our *intuitive model* of what such proofs should mean. These assumptions, together with the decidability of the form calculus, enable PegaSys to fully mechanize consistency proofs.

Reuse of a hierarchy

Having completed the host (its design, implementation, and verification), we would like to reuse it four times in Figure 5 with minimal reduplication of previous work. Below, we refer to the presently active development as the *primary* one and the development that we intend to reuse as the *secondary* one. At this stage of our example, the network is primary and the host secondary. We first consider reuse of the host design then its implementation.

There is a simple, yet useful, way to connect primary and secondary designs. An *atomic* active entity in a primary hierarchy may be replaced by an entire secondary hierarchy provided that (1) the atomic active entities that serve as interface to the secondary hierarchy are "matched up" with active entities of the primary hierarchy and (2) interactions with the replaced entity of the primary design are preserved (in the same sense as with active refinements).

This procedure is illustrated in Figure 14. The top window shows the lowest level of the network hierarchy, and the bottom one shows the highest level of the host hierarchy. We want to replace each of the host entities in the network (which are atomic) with the entire host hierarchy. Recall from Figure 6d that a host interfaces with a network backbone by means of the atomic Snd and Rcv entities. The Snd and Rcv entities of the host are associated (by pointing) with Snd and Rcv of the network, respectively. This pairing is done four times, once for each reuse of the host. In Figure 14, the user has started the series of pairings by selecting the leftmost Snd subpr[illegible] the network and the Snd subprogram of the host interface.* Figure 15 shows the final result. A double-ringed ellipse has been drawn around the network hosts to indicate their connection to another design hierarchy.

Reusability of an implementation is achieved by direct sharing of interface

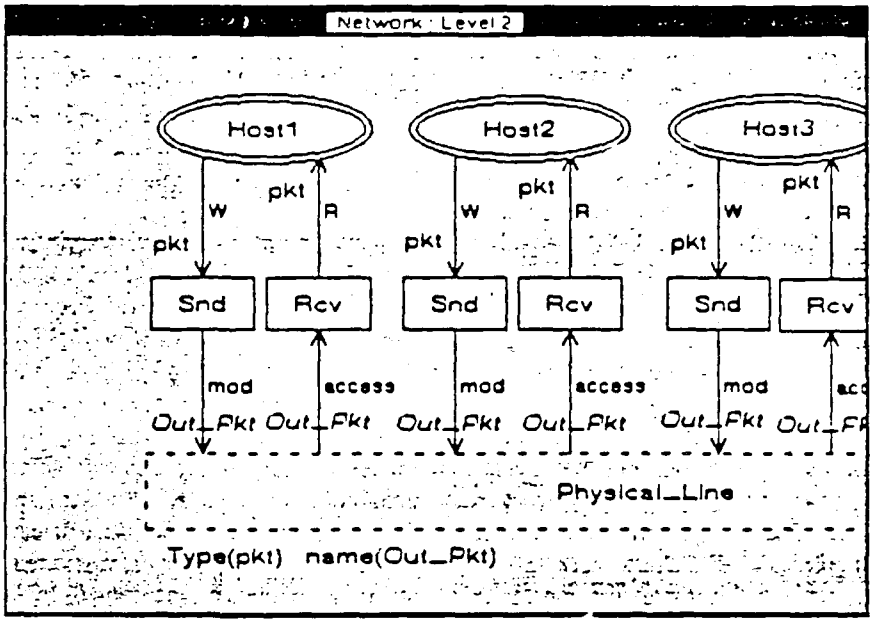


Figure 15. Hosts are marked to indicate reuse of an extant development.

*Things do not always work out as fortuitously as in this example. In particular, interfaces between designs do not always have identical interactions. In this event, it is sometimes possible to introduce a "dummy" entity that serves as an interface between the two designs.

code. For example, the implementation of Snd for the host must be identical, possibly with renaming, to the implementation of Snd for the network.* This is not as restrictive as it might sound. Most often, the interface of the secondary reusable implementation consists of code skeletons involving only headers for program units needed in the verification of the secondary design. Note that, under this condition, reverification of the secondary implementation is unnecessary.

At this point, the reader should not be misled into thinking that PegaSys always avoids unnecessary work. In fact, PegaSys presently is not incremental and duplicates work in many commonly occurring situations. In this example, the secondary host design and implementation are reused *before* the network has been implemented and verified. PegaSys would have to reverify the entire network (except for the host) if the network had been verified before reuse of the host. Our first priority has been to develop the basic capabilities of PegaSys, and we are now beginning to consider the problems of incremental analysis.

Having completed the host and reused it in the development of the network, the remaining task is to design and implement the physical line module. As the rest of the session follows the pattern of development already described, we omit it here.

PegaSys is an experimental system that we plan to extend in a number of ways. One area in which it is presently lacking involves the representation of persistent data and data dependencies, both of which arise in database applications. We expect to add several new capabilities, such as incremental analysis of changes, a sophisticated view mechanism, and a dynamic animation and testing facility. Animations would display particular execution states in terms of hierarchical pictures of the program design. In Figure 5, for example, we might show a packet flow from a host to the line whenever the "write" relation is satisfied.

While the precision and descriptive capability of pictures has legitimately been questioned in the past, PegaSys seems to suggest that it is possible to profitably combine both graphics and logic for a rich domain. Our limited experience suggests that PegaSys makes techniques more palatable to program developers. It is our expectation that such uses of graphics will lead to the utilization of formal documentation and analysis techniques by a wide, possibly mathematically unsophisticated audience. □

*The present implementation requires identical names.

Acknowledgment

This research was supported by the Office of Naval Research under Contract N00014-83-C-0300, by the Naval Electronics Systems Command under Contract N00039-82-C-0481, and by the Defense Advanced Research Projects Agency under Contract F30602-81-K-0176 (monitored by Rome Air Development Center).

References

1. "PegaSys: A Graphical Program Design Environment, Three Papers," tech. report CSL-145, Computer Science Laboratory, SRI International, June 1985.

2. M. Moriconi and D. F. Hare, "PegaSys: A System for Graphical Explanation of Program Designs," *Proc. ACM SIGPLAN 85 Symp. Language Issues in Programming Environments*, Seattle, Washington, June 1985, pp. 148-160. Also in tech. report CSL-145, Computer Science Laboratory, SRI International, June 1985.

3. M. Moriconi, "A Logical Basis for Graphical Description of Program Designs," tech. report CSL-145, Computer Science Laboratory, SRI International, June 1985. Submitted for publication.

4. I. Nassi and B. Shneiderman, "Flowchart Techniques for Structured Programming," *SIGPLAN Notices*, Vol. 8, No. 8, Aug. 1973, pp. 12-26.

5. E. Yourdan and L. L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1979.

6. M. L. Powell and M. A. Linton, "Visual Abstraction in an Interactive Programming Environment," *Proc. ACM SIGPLAN 83 Symp. Programming Language Issues in Software Systems*, June 1983, pp. 14-21.

7. A. L. Davis and R. M. Keller, "Data Flow Program Graphs," *IEEE Computer*, Vol. 15, No. 2, Feb. 1982, pp. 26-41.

8. C. Rich and H. Shrobe, "Initial Report on a Lisp Programmer's Apprentice," *IEEE Trans. Software Eng.*, Vol. SE-4, No. 6, Nov. 1978, pp. 456-466.

9. L. W. Cooprider, "The Representation of Families of Software Systems," PhD thesis, Computer Science Dept., Carnegie-Mellon University, April 1979.

10. F. DeRemer and H. H. Kron, "Programming-in-the-large Versus Programming-in-the-small," *IEEE Trans. Software Engineering*, Vol. SE-2, No. 2, June 1976, pp. 80-86.

11. H. C. Lauer and E. H. Satterthwaite, "The Impact of Mesa on System Design," *Proc. 4th Int'l Conf. Software Engineering*, Sept. 1979, pp. 174-182.

12. W. F. Tichy, "Software Development Control Based on Module Interconnection," PhD thesis, Computer Science Dept., Carnegie-Mellon University, Jan. 1980.

13. R. L. London, "Perspectives on Program Verification," R. T. Yeh (ed.), in *Current Trends in Programming Methodology*, Vol. 2, pp. 151-172, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1977.

14. S. P. Reiss, "Graphical Program Development with PECAN Program Development Systems," *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symp. Practical Software Development Environments*, April 1984, pp. 30-41.

15. S. P. Reiss, "PECAN: Program Development Systems that Support Multiple Views," *IEEE Trans. Software Eng.*, Vol. SE-11, No. 3, March 1985, pp. 30-41.

16. W. Swartout, "The Gist Behavior Explainer," *Proc. 3rd Nat'l Conf. Artificial Intelligence*, Aug. 1983, pp. 402-407.

17. M. H. Brown and R. Sedgewick, "Techniques for Algorithm Animation," *IEEE Software*, Vol. 2, No. 1, Jan. 1985, pp. 28-39.

18. R. Baecker, *Sorting Out Sorting*, 16mm color sound film, 25 min., 1981. (SIGGRAPH 1981, Dallas, Texas.)

19. R. M. Balzer, "EXDAMS—Extendable Debugging and Monitoring System," *Proc. AFIPS Spring Joint Computer Conf.*, 1969, pp. 567-580.

20. H. Lieberman, "Seeing What Your Programs Are Doing," tech. report 656, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Feb. 1982.

21. R. L. London and R. A. Duisberg, "Animating Programs Using Smalltalk," tech. report CR-84-30, Computer Research Laboratory, Tektronix, Inc., Beaverton, OR, Dec. 1984.

22. B. A. Myers, "Displaying Data Structures for Interactive Debugging," tech. report CSL-80-7, Xerox Palo Alto Research Center, Palo Alto, CA, June 1980.

23. A. Borning, "The Programming Language Aspects of ThinkLab, a Constraint-oriented Simulation Laboratory," *ACM Trans. Programming Languages and Systems*, Vol. 3, No. 4, Oct. 1981, pp. 353-387.

24. H. P. Frei, D. L. Weller, and R. Williams, "A Graphics-based Programming-support System," *Computer Graphics*, Vol. 12, No. 3, Aug. 1978, pp. 43-49.

25. R. V. Rubin, E. J. Golin, and S. P. Reiss, "ThinkPad: A Graphical System for Programming by Demonstration," *IEEE Software*, Vol. 2, No. 2, March 1985, pp. 73-79.

26. I. E. Sutherland, "SKETCHPAD: A Man-Machine Graphical Communication System," tech. report 296, MIT Lincoln Laboratory, Jan. 1963.

27. M. Wolfberg, "Fundamentals of the AMBIT/L List-processing Language," *SIGPLAN Notices*, Vol. 7, No. 10, Oct. 1972, pp. 66-76.

28. W. Teitelman, "A Tour Through Cedar," *IEEE Software*, Vol. 1, No. 2, April 1984, pp. 44-73.

29. *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A-1983, Dept. of Defense, US Govt. Print. Off., Washington, D.C., Jan. 1983.

30. L. Tesler, "The Smalltalk Environment," *BYTE*, Vol. 6, No. 8, Aug. 1981, pp. 90-146.

Mark Moriconi is a program director in the Computer Science Laboratory at SRI International. His research interests span the fields traditionally called "programming environments" and "program verification." He is especially interested in the construction of systems that effectively support and rigorously analyze software life-cycle activities. Moriconi's research activities have included the development of program verification systems, novel verification system components for analyzing incremental changes and for generating axiomatic semantic proofs from suitable tables, and, more recently, the PegaSys environment that accepts and reasons about pictures as formal documentation.

Moriconi holds a PhD degree in computer science from the University of Texas at Austin.

Dwight Hare is a researcher in the Computer Science Laboratory at SRI International. He has been involved in the design and construction of programming environments that involve the analysis of programs. He has designed and built a verification system using HDM (Hierarchical Development Methodology), which was used to prove properties of a fault tolerant operating system. He has contributed to the design and has been primarily responsible for the implementation of the PegaSys system.

Hare has a Master's degree in computer science from the University of Texas at Austin.

Questions about this article can be addressed to either author at the Computer Science Laboratory, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025.

SRI International

Reasoning About Design Changes

Mark Moriconi and Gail A. Harrison

SRI-CSL-88-14, November 1988
SRI Project 5912

Computer Science Laboratory
SRI INTERNATIONAL

333 Ravenswood Ave • Menlo Park, CA 94025
(415) 326-6200 • TWX: 910-373-2046 • Telex: 334-486

Reasoning About Design Changes

Mark Moriconi* Gail A. Harrison†
SRI International University of Washington

December 16, 1988

Abstract

A logical technique is presented for approximating the semantic effects of a change to a software system. The new method uses proofs about a system's *structure* to obtain after finitely many steps results that are reasonably close to optimal. The technique is general enough to apply to implementations and to formal specifications of abstract design structures, provided that abstractions are defined in terms of certain predefined primitives. An experimental implementation has been completed in Prolog.

1 Introduction

Programmers are continually faced with the problem of modifying an existing or partially developed system. Modifications must be made for several reasons: to correct erroneous behavior, to increase functionality, to adapt to a new environment, and to improve a correct implementation. A major difficulty in modifying a system, especially a large one, is that a seemingly minor change to one part of the system can have unforeseen and subtle consequences in another part.. As a result, the incremental cost of modifications is often unacceptably high.

In this paper, we formalize the basic question of whether a change to one system object can affect another system object, and we present a logical technique for

*Supported by the Naval Ocean Systems Center under Office of Naval Research contract N00014–86–C–0775 and by the Office of Naval Research under contract N00014–83–C–0300. Author's current address: Computer Science Laboratory (BN176), SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025.

†Research performed while visiting SRI under Office of Naval Research contract N00014–86–C–0775. Author's current address: Computer Science Department, University of Washington, Seattle, WA 98195.

answering a class of questions that can be reduced to instances of this basic question. Our approach has the following important properties:

- Potential changes are analyzed with respect to any level in system's design, whether the level contains abstract or concrete objects and connections. Early feedback on the effects of changes is provided whenever design decisions are formalized in a certain formal language.

- The semantic effects of a change are inferred from *structural* properties of a system. Our approach does not depend upon the presence of behavioral specifications, which can be difficult to construct. However, it does depend on the presence of structural information, which can be recorded explicitly by a programmer or extracted mechanically from the system implementation.

- A good approximation of the objects affected by a change can be found in a finite number of proof steps. In practice, this means that proofs can be fully mechanized, which will make it much easier to integrate our formal approach into everyday software development.

We are aware of no other approach to the analysis of changes that is general, rigorous, and fully mechanizable.

In this paper, a *design* is a logical description of how a system is put together, i.e., its interesting objects and the relevant structural relationships among them. This view of design focuses on the transition from a (formal) specification of *what* a system is intended to do to an implementation that describes *how* to perform the specified computation. This what-to-how transformation should be recorded explicitly because an implementation, especially an efficient one, will invariably be highly interconnected and somewhat unstructured. Designs will tend to be more modular and easier to understand, since performance is not as much of a concern.

Each level in a design can contain *user-defined relations* that denote abstractions of concrete objects and connections. Typically, an abstraction can be realized in multiple ways. For instance, "operation" is an abstract object that might be realized by a concrete procedure or function. The abstract relation "connected to" might be realized be various combinations of concrete procedure calls and data accesses and modifications. The lowest level in a design hierarchy directly models the structure of the implementation; this level can be derived from the implementation using classical flow analysis techniques. A design can be shown to be logically consistent with an implementation using the technique in [12].

It is undecidable in general to determine the exact behavioral effects of a change, but it is possible to obtain a precise, conservative approximation by formalizing the problem in terms of structural concepts. In particular, we say that a change to

an object x affects an object y if the pair $\langle x, y \rangle$ is in the transitive closure of the "information flow" relation. Unfortunately, information flow is *not* transitive in the usual sense. That is, if there is flow from some object x to an object y and from y to an object z, there is not necessarily flow from x to z. As a consequence, the usual notion of transitivity gives a crude approximation of the effects of a change. To obtain a more accurate approximation of the true transitive flows (i.e., those that would occur when the system is executed), we decompose the concept of information flow into three special flows and provide axioms for composing the three flows to determine the transitive closure of the information flow relation.

Since the effects of changes are analyzed in terms of information flows, the flows that are implicit in a structural specification must be enumerated. For example, if a concept in a specification is defined in terms of the primitive *mod(P,x)*, which says that variable x is modified by execution of procedure P, we know that there must be some variable v whose value is assigned to x. Hence, there is an implicit flow from v to x. Axioms are defined for deriving implicit flows from specifications written in our language. To substantially reduce the size of a specification, system interactions that are not intended to occur need not be specified (implicitly or explicitly). These interactions are dealt with by means of the closed-world assumption [13].

All axioms, specifications, and questions about changes are represented in a first-order logic restricted to finite models. Consequently, the proofs required to answer questions contain no infinite search paths. This is a direct consequence of the decision to take a structural approximation of a semantic property. The same decision has led to convergent algorithms for program flow analysis in the field of program optimization [1].

The remainder of this paper is organized as follows. The next section illustrates the problem that is the focus of the paper and defines it more precisely; the following section describes related work. Our solution is formalized in Sections 4–7, which present, respectively, our system model, the axioms for inferring the flows implicit in structural descriptions, our axiomatization of the transitivity of information flow, and the logical framework for stating and answering questions about the effects of changes. Section 8 summarizes our results.

2 Statement of the Problem

To illustrate the problem we address, consider the low-level design presented in Figure 1a. The design consists of several objects: procedures *AddInc*, *Add*, and *Inc*; variables *sum*, *i*, *a*, *b*, and *z*; and the constant 1. Parameters are transmitted using a value-result semantics. The purpose of procedure *AddInc*, which is not specified in the figure, is to add the initial values of *i* and *sum* and return the result

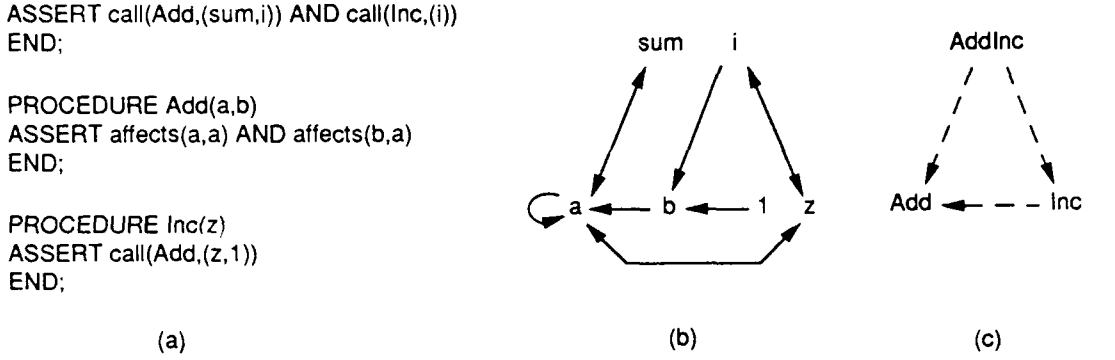


Figure 1: A low-level design: (a) textual representation, (b) diagram of implicit information flows, and (c) graph of call relationships.

in *sum*; it also increments the initial value of i and returns the result in i.

We are interested, for instance, in whether or not a change to the value of variable *sum* can affect the value of variable z. At first glance, there appears to be a relatively simple way to formalize this question in terms of the concept of "information flow." Informally, we say that there is information flow from variable x to variable y, denoted $x \Longrightarrow y$, if some change in the value of x affects the value of y when the program is executed.[1] For example, execution of the assignment statement "$b := a$" causes flow from a to b. Since two objects can interact indirectly through any number of intermediaries, the question is not whether $sum \Longrightarrow z$, but instead it is whether the pair $\langle sum, z \rangle$ is in the transitive closure of $\Longrightarrow$ on the set of all variables, written $sum \overset{*}{\Longrightarrow} z$.[2]

We cannot form the transitive closure of $\Longrightarrow$ until the information flow relationships implicit in the *assert* statements are made explicit. The assertion for *AddInc* says that it makes two calls, one to *Add* and one to *Inc*. (The ordering of the calls is unspecified for the moment.) The assertion for *Add* says that the initial values of a and b affect some future value of a. The assertion of *Inc* specifies that it calls

[1] Classical information theory, developed by Shannon [15] and others, is concerned with the *amount* of information generated by a particular event. We are interested in the simpler qualitative question of whether *any* information is generated by an event. In other words, we are interested in whether a change affects an object at all, not in how much it affects it.

[2] Let S be a set and R a relation on S. Relation R is *transitive* if "aRb and bRc" implies "aRc" for a, b, and c in S. Elements a, b, and c need not be distinct. The *transitive closure* of a relation R on a set S will be denoted R^*. We say that aR^*b if there exists a sequence $s_1, s_2, \ldots, s_n$ of zero or more elements in S such that $aRs_1, s_1Rs_2, \ldots, s_{n-1}Rs_n, s_nRb$.

Add.

Figure 1b depicts the information flow relationships implicit in this structural description. Flows not in the figure are assumed to be invalid, due to the closed-world assumption discussed later. For example, it is assumed that there is no flow in *Add* from a to b, making b a read-only variable of *Add*. Procedure calls normally cause bidirectional flow between actual and formal parameters. However, the two calls to *Add* cause only unidirectional flow from actual parameters i and 1 to formal parameter b, since the value of b is unchanged by *Add*.

Returning to our original question about the possibility of flow from *sum* to z, we can use the diagram Figure 1b to trace the information flow path

$$sum \Longrightarrow a \Longrightarrow z \tag{1}$$

from which we can infer that $sum \stackrel{*}{\Longrightarrow} z$. This kind of reasoning can be formalized in terms of the usual transitivity axiom.

Unfortunately, this simple analysis is much too conservative. A change to the value of *sum* cannot affect z if there is no execution sequence for which $sum \Longrightarrow z$. We can establish that there is no such sequence in our example specification by relating the information flow relationships in the specification to its calling relationships, which are illustrated in Figure 1c. Consider the call from *AddInc* to *Add*. Procedure *Add* contains no procedure calls and it does not allow the value of formal a to affect the value of its other formal b. Therefore, the call from *AddInc* to *Add* can only affect the value of *sum* by means of the path

$$sum \Longrightarrow a \Longrightarrow sum$$

Procedure *AddInc* also calls *Inc* with actual parameter i. But since i is never affected by *sum* (by the closed-world assumption), the call from *AddInc* to *Inc* cannot result in a flow from *sum* to z, from which we can conclude that the call from *Inc* to *Add* cannot either. This completes an informal argument that $\neg(sum \stackrel{*}{\Longrightarrow} z)$.

The main purpose of this paper is to formalize this kind of reasoning. To do so, a new axiomatization of the transitivity of information flow is required in which a transitive flow is inferred from two individual flows only when there is a *causal relationship* between the individual flows. That is, we must establish that some change in the value of variable x in the flow $x \Longrightarrow y$ causes a change in the value of variable z in the flow $y \Longrightarrow z$ before we can infer the transitive flow $x \Longrightarrow z$. Let $\mathcal{T}$ denote a logical axiomatization of transitive information flow that takes into account the causal relationships among individual flows. In addition, let $\mathcal{S}$ denote a structural specification, and let $\mathcal{I}$ denote axioms for inferring the flows implicit in specifications. To reason about changes, we must define $\mathcal{T}$ and $\mathcal{I}$ so that the transitive closure of $\Longrightarrow$, namely,

$$\{\langle x, y\rangle |\ \mathcal{T} \cup \mathcal{I} \cup \mathcal{S} \vdash x \Longrightarrow y\},$$

contains the true information flows in S. Moreover, the derivation of the closure should terminate for any specification S. The transitive closure with respect to a given specification S serves as the basis for answering a class of questions about changes to S.

3 Related Work

In practice, the effects of changes are usually determined informally. For example, a programmer might try to find the effects of a change by studying various relations extracted from the program itself, such as direct (cross-reference) relations and transitive relations about calling relationships and data flows. Numerous tools have been developed which derive such relations, but they are used primarily for other purposes, such as program optimization [3], the detection of simple errors [4], and documentation [10]. We are not aware of any existing tool that provides a systematic way of combining the relations to determine the effects of changes.

Moriconi [11] proposed a formal approach to the analysis of changes that requires the proof of formulas in an undecidable theory. From a behavioral specification of a system and a Hoare-style logic, his method determines what formulas must be proved to isolate the *exact* semantic effects of incremental changes. The drawback of this approach is that the effects of most changes cannot be found without substantial human assistance in proofs. The approach presented in this paper sacrifices exactness but has the important advantage of logical simplicity.

Recent work by Horwitz, Prins, and Reps [7] on interprocedural program slicing addresses the related problem of finding all program objects that might affect a distinguished object. They improve on the interprocedural slicing algorithm of Weiser [18] by defining a structural approximation of a slice in terms of operations on an extended program dependence graph. The problem of computing slices is an instance of the more general problem addressed in this paper. That is, our logical framework can be used to compute slices involving various kinds of concrete and abstract objects. For example, the variables x that can affect a specific variable v is

$$\{\langle x, v\rangle |\ \mathcal{T} \cup \mathcal{I} \cup \mathcal{S} \vdash x \Longrightarrow v\},$$

a subset of the closure discussed earlier. If we are interested in program slices, S consists of those instances of our primitive that are true of the program. A slice computed using our logical system appears to be as precise as one computed using the improved algorithm of Horwitz, et al.

Somewhat related is work on configuration management (e.g., [5,17]) that uses generic dependency relations between compilation units to determine what units must be recompiled following a change. The dependency relations are not powerful

enough for system design and a semantic property is not maintained. The goal is simply to ensure that a system is properly compiled.

4 System Model

Our primitives have been designed for describing a system that consists of a collection of procedures which communicate by parameters that are passed by value-result (copy-in/copy-out). A value-result semantics precludes the possibility of aliasing. We treat only scalar types and assume that objects (procedures, variables, and constants) have unique names. Modelling additional features, such as structured types, aliasing, shared global variables,[3] and modules, will require some adaptation; however, we believe that our basic approach is applicable.

The formal system model presented in this section should be distinguished from a language for writing structural specifications. The system model can be seen as a common internal representation for a class of specification languages. To illustrate the difference, consider the *call* relation in the specification of Figure 1a. It takes as arguments the called procedure and the set of actual parameters, whereas the corresponding relation in the system model is more verbose, taking as arguments the calling procedure, the called procedure, the actual parameters, the formal parameters, and the actual-formal pairings. In this paper, we do not propose a structural specification language; instead, we concentrate on the underlying system model. Henceforth, a structural specification is taken to be a logical expression in the system model unless stated otherwise. The signature of each primitive in the model is contained in Figure 2.

4.1 Primitive Objects

The basic objects in a design are variables and procedures, which are of type *var* and *proc*, respectively. A special kind of variable, called a *version variable*, is used to represent a value of an ordinary variable. Every time the value of an ordinary variable is changed, a new version variable is introduced. A version variable has type *vvar*. The standard mathematical concepts of boolean, finite sets, and finite sequences are also predefined types.

The version variables that can be used to specify the values of a variable must be explicitly associated with that variable. To this end, a finite sequence of distinct version variables is associated with each ordinary variable. The elements of such a sequence represent the successive values of the associated variable. A sequence for

[3] In our present model, global variables can be represented as additional parameters to each procedure. However, an explicit representation of globals is preferable.

Primitive Types

var

vvar

proc

Context

context: name $\rightarrow$ type $\times$ formals $\times$ locals $\times$ versions

Primitive Predicates (context argument left implicit)

$\Longrightarrow_f$: proc $\times$ vvar $\times$ vvar $\rightarrow$ bool

$\Longrightarrow_b$: proc $\times$ vvar $\times$ vvar $\rightarrow$ bool

$\Longrightarrow_l$: proc $\times$ vvar $\times$ vvar $\rightarrow$ bool

callByVR: proc $\times$ proc $\times$ (vvar $\times$ var)^n $\rightarrow$ bool, $\quad n \geq 0$

mod: proc $\times$ var $\rightarrow$ bool

acc: proc $\times$ var $\rightarrow$ bool

Figure 2: System Model.

a variable x must contain one element for each relevant change to x. A specification need not distiguish among the different values of a variable. In this event, a single version variable is used to model all values. Failure to distinguish among the different values will result in a more conservative approximation of the effects of a change.

We use the sequence operations *first* and *last* to return the first and last element, respectively, of a finite sequence. We also use functions that, when supplied with an element of a sequence, return the preceding and the next elements in the sequence. For an element e of a finite sequence s, the functions

$$\text{next: vvar} \rightarrow \text{vvar}$$

and

$$\text{prev: vvar} \rightarrow \text{vvar}$$

are defined by

$$\begin{aligned} \text{prev}(\text{next}(e, s), s) &= e \\ \text{prev}(\text{first}(s)) &= \text{first}(s) \\ \text{next}(\text{last}(s)) &= \text{last}(s) \end{aligned}$$

Example 1 The need for version variables can be illustrated by returning to the design in Figure 1 and asking whether or not $z \stackrel{*}{\Longrightarrow} sum$. Intuitively, a change in

the value of z should not affect *sum* because i must be incremented *after* it is added to *sum*. This means that procedure *AddInc* should call *Add* before *Inc*. Version variables can be used to specify this ordering.

If order is left unspecified (as in Figure 1a) and *AddInc* calls *Add* before *Inc*, the call to *Add* gives

$$sum \Longrightarrow a \Longrightarrow sum$$
$$i \Longrightarrow b \Longrightarrow a \Longrightarrow sum$$

and the call to *Inc* gives

$$i \Longrightarrow z \Longrightarrow a \Longrightarrow z \Longrightarrow i$$
$$1 \Longrightarrow b \Longrightarrow a \Longrightarrow z \Longrightarrow i$$

from which we conclude that $\neg(z \overset{*}{\Longrightarrow} sum)$.[4] Reversing the order of calls gives

$$i \Longrightarrow z \Longrightarrow a \Longrightarrow z \Longrightarrow i \Longrightarrow b \Longrightarrow a \Longrightarrow sum \qquad (2)$$

which implies that $z \overset{*}{\Longrightarrow} sum$.

Figure 3 shows how version variables can be used to achieve the desired ordering of calls. The key is in the assertion for *AddInc*, where the same input value i_{in} is transmitted to both *Add* and *Inc*. The value i_{out} returned by *Inc* is different from i_{in} and therefore i_{out} can never be transmitted to *Add*. This is illustrated by the information flow diagram in Figure 3b in which path (2) cannot occur. We can trace a path from z to i_{out}, namely,

$$i_{in} \Longrightarrow z_{in} \Longrightarrow a_{in} \Longrightarrow a_{out} \Longrightarrow z_{out} \Longrightarrow i_{out}$$

but there is no path from i_{out} to i_{in} and, hence, no path to *sum*. □

4.2 Contexts

An object has a name, a type, and possibly certain other properties. For instance, a procedure object has a name, the type *proc*, and a set of formal parameters. Objects are modelled by a function called a *context*, which would be derived from declarations written in a structural specification language or a programming language. Contexts are used in the evaluation of logical expressions.

Formally, a *context* for a specification S is a function

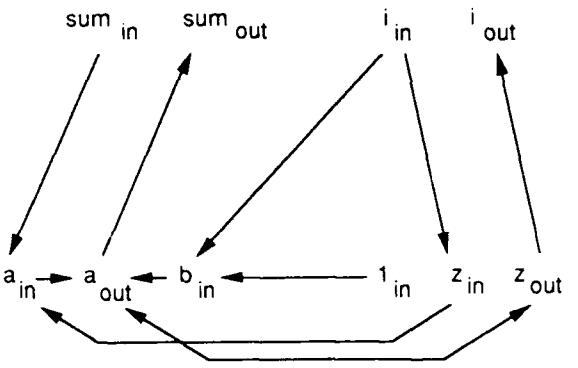
$$\text{context: names} \rightarrow \text{types} \times \text{formals} \times \text{locals} \times \text{versions}$$

where

[4] The path $i \Longrightarrow z \Longrightarrow a \Longrightarrow sum$ is not a possibility because there is no causal relationship between $i \Longrightarrow z \Longrightarrow a$ and $a \Longrightarrow sum$. This would be detected by the transitivity axioms of Section 6.

```
PROCEDURE AddInc(sum,i)
ASSERT call(Add,(sum_in,i_in)) AND
        call(Inc,(i_in))
END;

PROCEDURE Add(a,b)
ASSERT affects(a_in,a_out) AND
        affects(b_in,a_out)
END;

PROCEDURE Inc(z)
ASSERT call(Add,(z_in,1))
END;
```

(a)



(b)

Figure 3: Using version variables to order calls from *AddInc* to *Add* and *Inc*.

- *names* is a finite set containing the object names in S,
- *types* is a finite set containing the primitive and constructed types in S,
- *formals* is the finite powerset of the set of variables in S, used for recording the parameter lists of procedures,
- *locals* is the finite powerset of the set of variables in S, used for recording the local variables of procedures, and
- *versions* is a finite set whose elements are the version-variable sequences in S.

A context is fixed for a given specification. In this paper, the context associated with a specification S is treated as an *implicit argument* of every predicate in S, including the primitive predicates. For instance, let C be the context for the specification under consideration and let the function *allprocs*, when supplied with a context, return the set of all procedure names in it. Then, the expression $(\forall x\colon proc)$ is written as a shorthand for $(\forall x \in allprocs(C))$.

The following operations on contexts are used later in the paper. The predicate *versionOf* is a mapping $vvar \times var \rightarrow bool$ that checks whether a version variable is a version of an ordinary variable. The predicate *formalOf* (*localOf*) is a mapping $var \times proc \rightarrow bool$ that checks whether a variable is a formal (local) of a procedure. It is often convenient to ask whether a given variable or version variable is a variable of a procedure. The predicate

$$\text{varOf: } (\text{var} + \text{vvar}) \times \text{proc} \rightarrow \text{bool}$$

is defined by

$$\begin{aligned} \text{varOf}(x, P) &= \text{localOf}(x, P) \vee \text{formalOf}(x, P) \\ \text{varOf}(\hat{x}, P) &= (\exists x\text{: var})[\text{versionOf}(\hat{x}, x) \wedge \text{varOf}(x, P)] \end{aligned}$$

where the hat denotes a version variable. The only retrieval operation we will use is the function *versions*, which returns the version-variable sequence associated with a given variable.

Example 2 The declarations in the specification of Figure 3 define a context that can be represented in tabular form:

name	type	formals	locals	versions
$AddInc$	proc	$\emptyset$	$\{sum, i\}$	$\emptyset$
sum	var	$\emptyset$	$\emptyset$	$\langle sum_{in}, sum_{out} \rangle$
sum_{in}	vvar	$\emptyset$	$\emptyset$	$\emptyset$
sum_{out}	vvar	$\emptyset$	$\emptyset$	$\emptyset$
i	var	$\emptyset$	$\emptyset$	$\langle i_{in}, i_{out} \rangle$
i_{in}	vvar	$\emptyset$	$\emptyset$	$\emptyset$
i_{out}	vvar	$\emptyset$	$\emptyset$	$\emptyset$
Add	proc	$\{a, b\}$	$\emptyset$	$\emptyset$
a	var	$\emptyset$	$\emptyset$	$\langle a_{in}, a_{out} \rangle$
a_{in}	vvar	$\emptyset$	$\emptyset$	$\emptyset$
a_{out}	vvar	$\emptyset$	$\emptyset$	$\emptyset$
b	var	$\emptyset$	$\emptyset$	$\langle b_{in} \rangle$
b_{in}	vvar	$\emptyset$	$\emptyset$	$\emptyset$
Inc	proc	$\{z\}$	$\{1\}$	$\emptyset$
z	var	$\emptyset$	$\emptyset$	$\langle z_{in}, z_{out} \rangle$
z_{in}	vvar	$\emptyset$	$\emptyset$	$\emptyset$
z_{out}	vvar	$\emptyset$	$\emptyset$	$\emptyset$
1	var	$\emptyset$	$\emptyset$	$\langle 1_{in} \rangle$
1_{in}	vvar	$\emptyset$	$\emptyset$	$\emptyset$

Notice that every object has a name and a type. The other properties of an object depend on its type and the specification itself. A constant, such as the number 1, is modelled as a variable having exactly one version variable. □

4.3 Directed Information Flow

In our system model, information flows are associated with procedures. Informally, we say that information flows from a variable x to a variable y under procedure P

provided a change in the value of x can be conveyed to y when P is executed. For example, the binding of an actual parameter a to a formal parameter x causes a flow from a to x.

This concept can be formalized as follows. Let $store: id \rightarrow out$ model a computer memory as a mapping of identifiers to their values, and let $eval: proc \times store \rightarrow store$ be the valuation function for procedures. The infix operation . is a function $store \times id \rightarrow out$ that looks up the value of an identifier in a store. The equality predicate $s_1 \stackrel{x}{=} s_2$ determines whether or not two stores s_1 and s_2 have the same values for all identifiers except possibly for x. Information is transmitted from a to b by procedure P if and only if variety in a affects the value of b when P is executed. Formally,

$$a \stackrel{P}{\Longrightarrow} b \stackrel{\text{def}}{=} (\exists s_1, s_2)\ s_1 \stackrel{a}{=} s_2 \ \wedge\ eval(P, s_1).b \neq eval(P, s_2).b$$

where s_1, s_2 in $store$. This formulation of information transmission is a slight modification of the formulation originally developed by Cohen [2] for stating problems in computer security.

For our purposes, it is not enough to know that there is flow between two variables. In addition, we must know the "directionality" of the flow. Specifically, we define $\Longrightarrow$ to be the logical disjunction of three directed-flow relations: $\Longrightarrow_f$, $\Longrightarrow_b$, and $\Longrightarrow_l$, standing for forward, backward, and lateral information flow, respectively. Forward and backward flows model the interprocedural variable bindings that result from a direct or transitive procedure call. Lateral flow is intraprocedural, involving local variables of the same procedure. Henceforth, $x \Longrightarrow y$ is taken to mean that $\langle x, y \rangle$ is in any one of the three directed-flow relations. Formally, the relation

$$\Longrightarrow: proc \times vvar \times vvar \rightarrow bool$$

is defined by

$$x \stackrel{P}{\Longrightarrow} y \stackrel{\text{def}}{=} x \stackrel{P}{\Longrightarrow}_l y \ \vee\ x \stackrel{P}{\Longrightarrow}_f y \ \vee\ x \stackrel{P}{\Longrightarrow}_b y$$

Notice that for $a \stackrel{P}{\Longrightarrow} b$ to be true, neither a nor b can be a constant. For example, $3 \stackrel{P}{\Longrightarrow} b$ cannot be true since 3 contains no variety. Recall, however, that we decided to model a constant as a variable. More specifically, we model a constant as a *read-only variable*, i.e., a variable whose value cannot be changed by the program. In this model of constants, a question about the effects of an edit that would replace one constant with another is meaningful and can be answered without any additional machinery. A read-only variable must satisfy the derived *ReadOnly* predicate, which disallows flow to the variable, but allows flow to emanate from it.

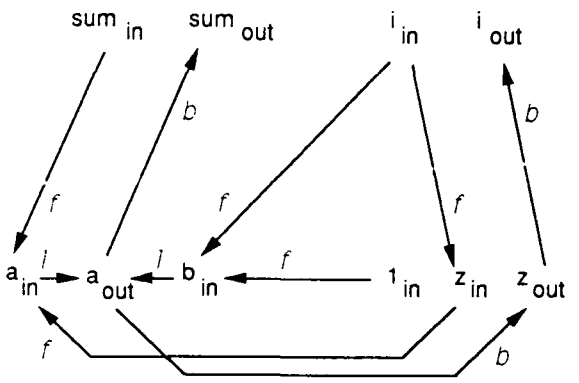


Figure 4: Directed information flows for the *AddInc* example in Figure 3a.

Example 3 The directionality of the flows seen earlier in Figure 3b is made explicit in Figure 4. For example, there is a forward flow from sum_{in} to a_{in} because the call from *AddInc* to *Add* causes the initial value of actual parameter *sum* to be bound to formal parameter *a* of *Add*. There is a backward flow from a_{out} to sum_{out} because the value of *a* is assigned to the output version of *sum* upon return of control from *Add* to *Inc*. □

4.4 Additional Primitive Connections

The n-ary *callByVR* relation is used to model procedure calls. Its first argument is a procedure P that directly calls a procedure Q with an arbitrary number of actual-formal parameter pairs. Each call has a value-result semantics and call chains can be circular. Sometimes we are not interested in the arguments of a call, in which case we use the function

$$\text{dcall: proc} \times \text{proc} \rightarrow \text{bool}$$

which is defined by

$$\text{dcall}(P, Q) = (\exists p\text{: } plist)\text{callByVR}(P, Q, p)$$

where *plist* is a set of pairs of type $vvar \times var$, which is the possible actual-formal pairings in a given context.

The *mod* and *acc* predicates are familiar in the field of program optimization [1]; they are also useful in building structural specifications. The relation $mod(P, x)$ says

that a variable x of procedure P (i.e., $x \in varOf(P)$) can be modified by execution of P, either directly or transitively through a called procedure.[5] The relation $acc(P, x)$ says that a variable x can be accessed by execution of procedure P. The *mod* and *acc* relations are not independent of the other primitive concepts; we will show how they can be defined in terms of flow relations.

4.5 Derived Abstractions

Ordinarily, a system design would not be specified directly in terms of the primitives. It instead would be expressed in terms of abstractions appropriate to each level of detail. Most abstract objects are represented naturally as primitive procedures or variables which are subsequently "implemented" in terms of one or more similar objects. On the other hand, most abstract dependencies are best represented as derived concepts defined in terms of more primitive dependencies. Abstract dependencies are used to partition a system into manageable parts that interact in well-defined and predictable ways. Several useful derived dependencies are defined below.[6]

Example 4 *Protecting a variable.* It is often useful to restrict access to a variable or to restrict the ways in which a variable can be used. For instance, we may want to allow procedures to read a certain variable but prohibit them from writing it. This is captured by the predicate

$$\text{ReadOnly}: \text{var} \rightarrow \text{bool}$$

which is defined by

$$\text{ReadOnly}(x) \stackrel{\text{def}}{=} \neg (\exists p: \text{proc}) \text{mod}(p, x)$$

for x in *var*. If a variable is required to satisfy this predicate, we can specify accesses of the variable, but any specified modification to it will be inconsistent with the above definition. □

Example 5 *Restricting variable interactions.* A set of variables can be partitioned into independent subsets using a predicate which says that a variable x is completely

[5] For optimization purposes, *mod* and *acc* usually contain only variables visible at the interface to a procedure. However, we must also include local variables not visible at the interface.

[6] Design dependencies can be stated informally using various program design languages, several of which are described in a book by Martin and McClure [9]. These languages provide a few useful primitive concepts, but they do not support definitional extensions and their meaning is imprecise and possibly ambiguous.

independent of a variable y if and only if a change in the value of y has no effect on the value of x. This predicate

$$\text{IndependentOf}: \text{var} \times \text{var} \rightarrow \text{bool}$$

is defined, for x and y in *var*, by

$$\begin{aligned}&\text{IndependentOf}(x, y) \stackrel{\text{def}}{=} \\ &\quad (\forall \hat{x}, \hat{y}: \text{vvar})(\forall R: \text{proc})[\text{versionOf}(\hat{x}, x) \wedge \text{versionOf}(\hat{y}, y) \supset \neg(\hat{y} \stackrel{R}{\Longrightarrow} \hat{x})]\end{aligned}$$

If a variable x is independent of a variable y, we know that y cannot use x as an intermediary to affect some other variable or procedure. □

Example 6 *Interprocedural channel.* Suppose that we want two procedures to communicate through a specific variable. We say that a variable x is a *channel* from procedure P to procedure Q iff information flows from P to Q through x. This is captured by

$$\text{ChannelTo}: \text{proc} \times \text{proc} \times \text{var} \rightarrow \text{bool}$$

which is defined by

$$\begin{aligned}&\text{ChannelTo}(P, Q, x) \stackrel{\text{def}}{=} \\ &\quad (\exists \hat{x}, \hat{y}, \hat{z}: \text{vvar})(\exists R: \text{proc})[\text{versionOf}(\hat{x}, x) \wedge \text{varOf}(\hat{y}, P) \wedge \text{varOf}(\hat{z}, Q) \wedge \\ &\qquad ((\hat{y} \stackrel{R}{\Longrightarrow}_f \hat{x} \wedge \hat{x} \stackrel{R}{\Longrightarrow}_f \hat{z}) \vee (\hat{y} \stackrel{R}{\Longrightarrow}_b \hat{x} \wedge \hat{x} \stackrel{R}{\Longrightarrow}_f \hat{z}) \vee (\hat{y} \stackrel{R}{\Longrightarrow}_b \hat{x} \wedge \hat{x} \stackrel{R}{\Longrightarrow}_b \hat{z}))]\end{aligned}$$

for P and Q in *proc* and x in *var*. Since x is an interprocedural channel, we need not consider lateral flows whose purpose is to link interprocedural flows. We also rule out the possibility of a forward-backward flow, since this would make x a channel from P to itself. □

Example 7 *Interprocedural partitioning.* Assume that a procedure A is not intended to be connected to a procedure B, which we express by

$$\neg\text{ConnectedTo}(A, B)$$

The *ConnectedTo* relation says that, for any procedures P and Q, there is a transitive call from P to Q, or a transitive information flow from a variable referenced by P to one referenced by Q, or both. The predicate

$$\text{Calls}: \text{proc} \times \text{proc} \rightarrow \text{bool}$$

is defined recursively by

$$\begin{aligned}&\mathrm{Calls}(P,Q) \stackrel{\mathrm{def}}{=} \\ &\quad (\exists p\colon \mathrm{plist})[\mathrm{callByVR}(P,Q,p) \vee \\ &\qquad (\exists R\colon \mathrm{proc})[\mathrm{callByVR}(P,R,p) \wedge \mathrm{Calls}(R,Q)]]\end{aligned}$$

where, as before, *plist* is a set of possible actual-formal pairings. The predicate

$$\mathrm{ConnectedTo}\colon \mathrm{proc} \times \mathrm{proc} \rightarrow \mathrm{bool}$$

is defined by

$$\begin{aligned}&\mathrm{ConnectedTo}(P,Q) \stackrel{\mathrm{def}}{=} \\ &\quad \mathrm{Calls}(P,Q) \ \vee \ (\exists \hat{x},\hat{y}\colon \mathrm{vvar})(\exists R\colon \mathrm{proc})[\mathrm{varOf}(\hat{x},P) \wedge \mathrm{varOf}(\hat{y},Q) \wedge \hat{x} \stackrel{R}{\Longrightarrow} \hat{y}]\end{aligned}$$

Notice that information may flow from P to Q as the result of a transitive call from P to Q (in which case R is P), or R can be a parent of P and Q that transmits a return flow from P to Q. □

5 Inferring Directed Flows

We must identify any implicit flows in a specification before we can apply our transitivity axioms. The axioms presented in this section can be used to deduce the flows left implicit in any specification constructed using the primitives.

The first two axioms in Figure 5 allow us to infer directed flows from calls. The VP axiom handles the situation in which the value of an actual parameter is actually used by the called procedure. In this event, there is a forward flow from the actual parameter to the corresponding formal parameter. In the antecedent of the axiom, *af* is an actual-formal pair and *afpairs* (of type *plist*) is a set of such pairs. The *member* operation tests whether *af* is in *afpairs*. In the consequent, the value of the expression *first(af)* is a version variable transmitted as an actual parameter; the value of *first(versions(last(af)))* is the input version variable for the corresponding formal parameter.

The RP axiom says that a backward flow from a formal to its corresponding actual occurs only when the value of the formal is modified during execution. If it is not, there is no need to return its value and, hence, no backward flow is necessary. The consequent of this axiom specifies a backward flow from the output version variable associated with the formal parameter in the pair *af* to the next version of the variable transmitted as an actual parameter.

Normally, there is a bidirectional flow between actuals and formals. The use of two separate axioms, however, will improve our estimate of the effects of a change

Value Parameter (VP)

$$\begin{aligned}&\text{callByVR}(P, Q, \mathit{afpairs}) \wedge \text{member}(\mathit{af}, \mathit{afpairs}) \wedge \text{acc}(Q, \text{last}(\mathit{af}))\\ &\quad \supset \quad \text{first}(\mathit{af}) \stackrel{P}{\Longrightarrow}_f \text{first}(\text{versions}(\text{last}(\mathit{af})))\end{aligned}$$

Result Parameter (RP)

$$\begin{aligned}&\text{callByVR}(P, Q, \mathit{afpairs}) \wedge \text{member}(\mathit{af}, \mathit{afpairs}) \wedge \text{mod}(Q, \text{last}(\mathit{af}))\\ &\quad \supset \quad \text{last}(\text{versions}(\text{last}(\mathit{af})) \stackrel{P}{\Longrightarrow}_b \text{next}(\text{first}(\mathit{af}))\end{aligned}$$

Mod and Information Flow (MI)

$$\begin{aligned}&\text{mod}(P, x) \quad \equiv\\ &\quad (\exists \hat{x}\text{: vvar})[\text{versionOf}(\hat{x}, x) \wedge\\ &\qquad (\exists \hat{y}\text{: vvar})[(\text{varOf}(\hat{y}, P) \wedge \hat{y} \stackrel{P}{\Longrightarrow}_l \hat{x}) \ \vee\\ &\qquad\qquad (\exists Q\text{: proc})[\text{dcall}(P, Q) \wedge \text{varOf}(\hat{y}, Q) \wedge \hat{y} \stackrel{P}{\Longrightarrow}_b \hat{x}]]]\end{aligned}$$

Acc and Information Flow (AI)

$$\begin{aligned}&\text{acc}(P, x) \quad \equiv\\ &\quad (\exists \hat{x}\text{: vvar})[\text{versionOf}(\hat{x}, x) \wedge\\ &\qquad (\exists \hat{y}\text{: vvar})[(\text{varOf}(\hat{y}, P) \wedge \hat{x} \stackrel{P}{\Longrightarrow}_l \hat{y}) \ \vee\\ &\qquad\qquad (\exists Q\text{: proc})[\text{dcall}(P, Q) \wedge \text{varOf}(\hat{y}, Q) \wedge \hat{x} \stackrel{P}{\Longrightarrow}_f \hat{y}]]]\end{aligned}$$

Figure 5: Finding implicit flows. The type of each free variable can be inferred from the signatures in Figure 2.

whenever the flow happens to be unidirectional. If a formal parameter is not modified by the called procedure, there is no return flow. If a formal parameter is used only to return values, there is no forward flow. If a formal parameter is not used at all, no flow occurs and none can be inferred using the axioms.

The axioms defining *mod* and *acc* are intuitively simple but syntactically complex. Axiom MI says that a variable x is modified by P if and only if a version of x is modified within P or as a result of a return flow from a called procedure. Axiom AI says that x is accessed by P if and only if a version of x is accessed within P or is transmitted by P as an actual parameter. Variable modification and access are determined by directed flows: if $a \stackrel{P}{\Longrightarrow} b$, then a is accessed and b is modified.

6 Transitivity Using Directed Flows

To accurately track the flow of information, we introduce twelve logical axioms in Figure 6 that define transitivity for the information flow relation. The axiom schema at the top of the figure says directional flows are transitive in the usual sense.

The next four axioms combine directional flows with lateral flows. Lateral flows among variables of a procedure are "directionless" in that they are used to link interprocedural flows and to allow interprocedural flow propagation to proceed in either direction. For instance, Axiom FL says that if a forward flow is followed by a lateral flow, the overall direction of flow is forward. The direction will stay forward unless it is changed by a backward flow. Similarly, axiom LF says that a lateral flow followed by a forward flow results in a forward flow. In both instances, the lateral flow serves as an intermediate flow connecting to a propagated forward flow.

The BF and FB axioms combine forward and backward flows. Axiom BF says that if there is a backward flow from a variable x in a called procedure to a variable y in its caller, and the caller then transmits y forward to variable z through another call, the resultant direction of flow from x to z is forward.

Axiom FB is somewhat complicated because it must trace a flow emanating from a call site to the called procedure and back to the *same* call site. This is accomplished by the third conjunct in the antecedent. This conjunct contains two disjuncts, which handle the two situations illustrated in Figure 7. In both instances, procedure P calls procedure Q and variable y belongs to Q. The first disjunct (Figure 7a) says that when the value of actual x is transmitted to formal a, Q may modify the value of a and then transmit this new value back to x. If this occurs, the next version of x, $next(x)$, is assigned the value. The input value of a_{in} can be transmitted directly or indirectly to a_{out}.

The second disjunct (Figure 7b) handles the case in which the value of x affects

Lateral-Lateral (LL), Forward-Forward (FF), Backward-Backward (BB)

$$x \overset{P}{\Longrightarrow}_\delta y \ \wedge \ y \overset{P}{\Longrightarrow}_\delta z \ \supset \ x \overset{P}{\Longrightarrow}_\delta z, \qquad \text{if } \delta \text{ is } l, f, \text{or } b.$$

Forward-Lateral (FL)

$$x \overset{P}{\Longrightarrow}_f y \ \wedge \ y \overset{P}{\Longrightarrow}_l z \ \supset \ x \overset{P}{\Longrightarrow}_f z$$

Lateral-Forward (LF)

$$x \overset{P}{\Longrightarrow}_l y \ \wedge \ y \overset{P}{\Longrightarrow}_f z \ \supset \ x \overset{P}{\Longrightarrow}_f z$$

Backward-Lateral (BL)

$$x \overset{P}{\Longrightarrow}_b y \ \wedge \ y \overset{P}{\Longrightarrow}_l z \ \supset \ x \overset{P}{\Longrightarrow}_b z$$

Lateral-Backward (LB)

$$x \overset{P}{\Longrightarrow}_l y \ \wedge \ y \overset{P}{\Longrightarrow}_b z \ \supset \ x \overset{P}{\Longrightarrow}_b z$$

Backward-Forward (BF)

$$x \overset{P}{\Longrightarrow}_b y \ \wedge \ y \overset{P}{\Longrightarrow}_f z \ \supset \ x \overset{P}{\Longrightarrow}_f z$$

Forward-Backward (FB)

$$\begin{aligned} & x \overset{P}{\Longrightarrow}_f y \ \wedge \ y \overset{P}{\Longrightarrow}_b z \ \wedge \ (\exists Q\text{:proc})(\exists a, b\text{:var})[\ \text{varOf}(y, Q) \ \wedge \\ & \qquad [\text{next}(x) = z \wedge \text{callByVR}(P, Q, \langle x, a\rangle) \ \wedge \ \text{first}(\text{versionsOf}(a)) \overset{Q}{\Longrightarrow}_l y \\ & \qquad\qquad \wedge \ (y \overset{Q}{\Longrightarrow}_l \text{last}(\text{versionsOf}(a)) \ \vee \ y = \text{last}(\text{versionsOf}(a)))] \ \vee \\ & \qquad [\text{sameCall}(P, Q, \{\langle x, a\rangle, \langle \text{prev}(z), b\rangle\}) \ \wedge \ \text{first}(\text{versionsOf}(a)) \overset{Q}{\Longrightarrow}_l y \\ & \qquad\qquad \wedge \ (y \overset{Q}{\Longrightarrow}_l \text{last}(\text{versionsOf}(b)) \ \vee \ y = \text{last}(\text{versionsOf}(b)))]] \\ \supset \ & x \overset{P}{\Longrightarrow}_l z \end{aligned}$$

Upward flattening (UF)

$$\text{dcall}(P, Q) \wedge x \overset{Q}{\Longrightarrow}_\delta y \ \supset \ x \overset{P}{\Longrightarrow}_\delta y, \qquad \text{if } \delta \text{ is } f, b, \text{or } l.$$

Figure 6: Transitivity of $\Longrightarrow$. Free variables of type *vvar* (version-variable) are indicated by a small letter, of type *proc* (abstract procedure) by a capital letter.

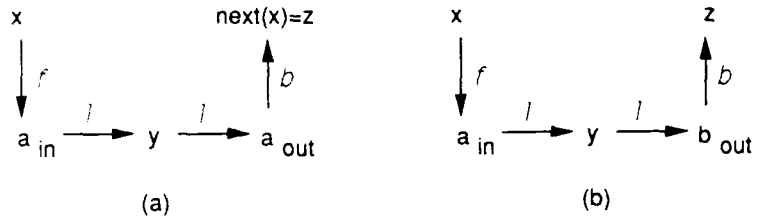


Figure 7: Illustration of the two situations handled by axiom FB.

the value returned to another actual parameter. The derived predicate *sameCall* is true if two actual-formal parameter pairs are associated with the same call site. In axiom FB,

$$\text{sameCall}(P, Q, \{\langle x, a\rangle, \langle \text{prev}(z), b\rangle\})$$

is true if actual x of P is associated with formal a of Q and the previous version of z (the one before the call) is associated with actual b. That is,

$$\text{sameCall: proc} \times \text{proc} \times \text{plist} \rightarrow \text{bool}$$

is defined by

$$\begin{aligned}\text{sameCall}(P, Q, p_1) = (\exists p_2\text{: plist})[&\text{callByVR}(P, Q, p_2) \wedge \\ &(\forall p\text{: ppair})(\text{member}(p, p_1) \supset \text{member}(p, p_2))]\end{aligned}$$

where *ppair* is of type *vvar* $\times$ *var*, an actual-formal pair.

If the premise of axiom FB is satisfied, there is a lateral flow from actual x to actual z. This has the effect of masking the procedure call, and it permits z to be propagated in a lateral, forward, or backward flow initiated by P.[7]

The UF axiom schema, in conjunction with the FB axiom, specifies when interprocedural flows can legitimately be combined. The UF schema allows flows to be combined only if they occur on control paths emanating from a common procedure. More precisely, the schema says that a flow resulting from the execution of procedure Q also results from the execution of procedure P provided P directly calls Q.

[7] Axiom FB can be stated more elegantly; the formulation presented in this section was chosen to mirror the structure of the other axioms as closely as possible.

Example 8 Returning to Figure 3a, suppose that we want to ask "Does the value of i affect the value of sum?" To answer this question, we must collect the declared objects into a context, which was done earlier in Example 2, and translate the specified connections into predicates in our logic. The translation gives

P1. $\text{callByVR}(AddInc, Add, \langle sum_{in}, a\rangle, \langle i_{in}, b\rangle)$
P2. $\text{callByVR}(AddInc, Inc, \langle i_{in}, z\rangle)$
P3. $a_{in} \overset{Add}{\Longrightarrow}_l a_{out}$
P4. $b_{in} \overset{Add}{\Longrightarrow}_l a_{out}$
P5. $\text{callByVR}(Inc, Add, \langle z_{in}, a\rangle, \langle 1_{in}, b\rangle)$

where the *call* and *affects* relations in the figure have been translated into the $callByVR$ and $\Longrightarrow_l$ relations, respectively. We now use the axioms in Figure 5 to deduce the following implicit flows; together with P3 and P4, they correspond to the ten arrows in Figure 4.

P6. $sum_{in} \overset{AddInc}{\Longrightarrow}_f a_{in}$
P7. $a_{out} \overset{AddInc}{\Longrightarrow}_b sum_{out}$
P8. $i_{in} \overset{AddInc}{\Longrightarrow}_f b_{in}$
P9. $i_{in} \overset{AddInc}{\Longrightarrow}_f z_{in}$
P10. $z_{out} \overset{AddInc}{\Longrightarrow}_b i_{out}$
P11. $z_{in} \overset{Inc}{\Longrightarrow}_f a_{in}$
P12. $a_{out} \overset{Inc}{\Longrightarrow}_b z_{out}$
P13. $1_{in} \overset{Inc}{\Longrightarrow}_f b_{in}$

Given assumptions P1–P13, the answer to our question is provided by the following formal proof.

1.	$dcall(AddInc, Add)$	premise P1, defn. of $dcall$
2.	$b_{in} \overset{AddInc}{\Longrightarrow}_l a_{out}$	UF(1, P4)
3.	$i_{in} \overset{AddInc}{\Longrightarrow}_f a_{out}$	FL(P8, 2)
4.	$\text{sameCall}(AddInc, Add, \{\langle i_{in}, b\rangle, \langle sum_{in}, a\rangle\})$	premise P1, defn. of $sameCall$
5.	$i_{in} \overset{AddInc}{\Longrightarrow}_l sum_{out}$	FB(3, P7, 4)

The application of the FB axiom in the last step is for the situation depicted in Figure 7b. □

Example 9 A slightly more difficult question is "Does the input value of i affect its output value?" The following proof illustrates how the lateral flow in the consequent of the FB axiom provides a neutral platform for propagating directional flows. Both applications of the FB axiom are for the situation in Figure 7a.

1.	$dcall(Inc, Add)$	P5, defn. of *dcall*
2.	$a_{in} \overset{Inc}{\Longrightarrow}_l a_{out}$	UF(1, P3)
3.	$z_{in} \overset{Inc}{\Longrightarrow}_f a_{out}$	FL(P11, 2)
4.	sameCall$(Inc, Add, \{\langle z_{in}, a\rangle\})$	P5, defn. of *sameCall*
5.	$z_{in} \overset{Inc}{\Longrightarrow}_l z_{out}$	FB(3, P12, 4)
6.	$dcall(AddInc, Inc)$	P2, defn. of *dcall*
7.	$z_{in} \overset{AddInc}{\Longrightarrow}_l z_{out}$	UF(5, 6)
8.	$i_{in} \overset{AddInc}{\Longrightarrow}_f z_{out}$	FL(P9, 7)
9.	sameCall$(AddInc, Inc, \{\langle i_{in}, z\rangle\})$	P2, defn. of *sameCall*
10.	$i_{in} \overset{AddInc}{\Longrightarrow}_l i_{out}$	FB(8, P10, 9)

□

7 Questions, Answers, and the Logic

The special flow axioms, structural specifications, and questions about changes are all represented in a single logic. Let DDB denote a "design data base" consisting of finitely many formulas that include a structural specification S, the transitivity axioms $\mathcal{T}$, and the rules I for inferring implicit flows, all expressed in or translated into the language $\mathcal{L}(DDB)$. $\mathcal{L}(DDB)$ is a typed (many-sorted) first-order logic with equality having the following properties:

1. There are a finite number of constant signs. The constants are pairwise distinct, and each one denotes a different design object, such as *AddInc* or *sum*.

2. The predicate signs are $\Longrightarrow_f$, $\Longrightarrow_b$, $\Longrightarrow_l$, *mod*, *acc*, and *callByVR*.

3. The type symbols are *var*, *vvar*, *proc*, *bool*, *seq*, and *set*.

4. The well-formed formulas are definite Horn clauses of the form

$$H_1 \wedge \cdots \wedge H_n \supset C, \qquad n \geq 0$$

where the H_i and C are atoms containing no function symbols.[8]

Definitions introduce new, eliminable symbols and we regard them as additional axioms. A *query* Q consists of

1. A declaration of the form $x_1{:}\,t_1, \ldots, x_n{:}\,t_n$, and

[8]The functions used in this paper are total and we know the values for any of their arguments. There are no Skolem functions because wffs are quantifier free.

2. An expression of the form

$$(q_1 y_1{:}\, T_1) \ldots (q_m y_m{:}\, T_m)\; W(x_1, \ldots, x_n, y_1, \ldots, y_m)$$

where $(q_i y_i{:}\, T_i)$ is $(\forall y_i{:}\, T_i)$ or $(\exists y_i{:}\, T_i)$, the t's and T's are types in $\mathcal{L}(DDB)$, and $W(x_1, \ldots, x_n, y_1, \ldots, y_m)$ is a quantifier-free formula in $\mathcal{L}(DDB)$ having free variables $x_1, \ldots, x_n$ and bound variables $y_1, \ldots, y_m$.

Before defining what it means for an n-tuple of constants to be an answer to a query, we should point out that it is not possible to deduce negative information with the inference system defined above. For instance, in our earlier analysis of the specification in Figure 3a, we concluded informally that $\neg(sum \stackrel{*}{\Longrightarrow} z)$. Intuitively, this is the correct answer. However, it is not possible to infer $\neg(sum \stackrel{*}{\Longrightarrow} z)$ from a DDB containing that specification.

Implicit in our informal reasoning was the assumption that all intended flows were specified and that those that were not specified could not occur. This assumption must be removed or it must be taken into account in the inference system. The former approach requires that all relevant positive and negative facts about the system be stated explicitly in the specification. In our domain, the number of negative facts can far exceed the number of positive ones, making it impractical to include the negative facts in a specification. The alternative approach is to specify all positive facts explicitly and modify a traditional first-order inference system to infer negative facts by default.[9] This can be formalized in terms of Reiter's *closed-world assumption* (CWA) [13], which says that given a data base DB and an atom A, if $DB \nvdash A$, then we can infer $\neg A$. Formally, the *CWA closure* of a DB is defined by

$$\text{closure}(DB) = DB \cup \{\neg P(\bar{c}) |\; DB \nvdash P(\bar{c})\}$$

where each $P(\bar{c})$ is a ground atomic formula. The CWA closure is known to be consistent for definite Horn clauses [16], and any positive or negative atomic query can be evaluated with respect to it.

An answer to a query can now be defined as follows. An n-tuple of constants $c_1, \ldots, c_n$ is an *answer* to a query Q with respect to DDB iff

1. $c_1 \in t_1, \ldots, c_n \in t_n$, and
2. $\text{closure}(DDB) \vdash (q_1 y_1{:}\, T_1) \ldots (q_m y_m{:}\, T_m)\; W(c_1, \ldots, c_n, y_1, \ldots, y_m)$

It is clearly *decidable* whether or not an n-tuple of constants is an answer to a query, since there are only finitely many constants and predicates in the intended

[9] Negative facts may be included in a structural specification for expository purposes, but they can be removed from the DDB because they have no influence on CWA query evaluation.

interpretation. However, if a design contains a large number of objects, it may be impractical to evaluate a query using a brute-force approach. The major cause of inefficiency is the recursive nature of the information-flow axioms. One way to eliminate the possibility of infinite deductions is to represent certain information flow relationships explicitly as ground atoms, rather than intentionally as general facts [14]. But this can require an excessive amount of storage if ground atoms are stored in the obvious way, e.g., as a set or array of atoms. An open problem is the development of a time and space efficient decision procedure for computing the transitive closure of $\Longrightarrow$ in accordance with our axioms.

The following examples illustrate how to represent questions about variables and procedures in our logic.

Example 10 *Variables.* Suppose we are interested in those variables x that are affected by a change to a given variable v of procedure P. Formally, we can express this by

$$(\exists \hat{x}, \hat{v}\colon \text{vvar})(\exists R\colon \text{proc})$$
$$[\text{versionOf}(\hat{x}, x) \wedge \text{varOf}(v, P) \wedge \text{versionOf}(\hat{v}, v) \wedge \hat{v} \stackrel{R}{\Longrightarrow} \hat{x}]$$

The variable x is the only free variable; v and P are logical constants. The formula says that a variable x is affected by a change to a variable v (of P) if a change to a version of v can affect a version of x.

Our earlier question about whether *sum* affects z is an instance of this question. If we substitute *sum* for v, *AddInc* for P, and z for x, we obtain

$$(\exists \hat{x}, \hat{v}\colon \text{vvar})(\exists R\colon \text{proc})$$
$$[\text{versionOf}(\hat{x}, z) \wedge \text{varOf}(sum, AddInc) \wedge \text{versionOf}(\hat{v}, sum) \wedge \hat{v} \stackrel{R}{\Longrightarrow} \hat{x}]$$

which is a formal statement of the question. This formula is not entailed by the closure of the *DDB* containing the specification in Figure 3. Therefore, by the CWA, we can conclude that *sum* does not affect z. □

Example 11 *Procedures.* Questions about procedures can be reduced to questions about variables. For instance, if we want to know each procedure Q affected by a change to a given variable v of procedure P, we write

$$(\exists \hat{x}, \hat{v}\colon \text{vvar})(\exists R\colon \text{proc})$$
$$[\text{varOf}(\hat{x}, Q) \wedge \text{varOf}(v, P) \wedge \text{versionOf}(\hat{v}, v) \wedge \hat{v} \stackrel{R}{\Longrightarrow} \hat{x}]$$

This expression says that a procedure Q is affected by a change to v if it has a variable that is affected by a change to v.

Similarly, the question of which variables x can be affected by a change to given procedure P is expressed by

$$(\exists \hat{x}, \hat{v}\colon \text{vvar})(\exists R\colon \text{proc})[\text{versionOf}(\hat{x}, x) \wedge \text{varOf}(\hat{v}, P) \wedge \hat{v} \stackrel{R}{\Longrightarrow} \hat{x}]$$

and the question of which procedures Q affected by a change to a given procedure P by

$$(\exists \hat{x}, \hat{v}\colon \text{vvar})(\exists R\colon \text{proc})[\text{varOf}(\hat{x}, Q) \wedge \text{varOf}(\hat{v}, P) \wedge \hat{v} \stackrel{R}{\Longrightarrow} \hat{x}]$$

□

Example 12 *Abstract slice.* An abstract slice is those variables x that can affect a given variable v of procedure P, which is the converse of the previous questions. This concept can be expressed as

$$(\exists \hat{x}, \hat{v}\colon \text{vvar})(\exists R\colon \text{proc})$$
$$[\text{versionOf}(\hat{x}, x) \wedge \text{varOf}(v, P) \wedge \text{versionOf}(\hat{v}, v) \wedge \hat{x} \stackrel{R}{\Rightarrow} \hat{v}]$$

The usual notion of a slice is concerned with the individual statements where variables are affected. We have chosen procedures, not statements, as atomic objects to facilitate the design and debugging of large-scale systems. Statement-level objects are more suited to program merging, for example, which is an important application of slicing [6]. □

The logical system and the question-answering technique defined in this section have been implemented in a version of Prolog that employs the *negation as failure* inference rule to infer negative information. A Prolog program is a set of definite Horn clauses that are executed using a refinement of the resolution principle called SLD resolution. A branch in an SLD proof tree is called a *success branch* if the derivation of the goal succeeds and a *failure branch* if it fails. A *finitely failed* SLD tree is one which is finite and contains no success branches. The negation as failure rule says that if an atom A has a finitely failed SLD tree for a given DDB, then infer $\neg A$ from that DDB. Since the SLD finite failure set is a subset of the complement of the success set, the negation as failure rule is less powerful that the CWA. Nevertheless, it is used for inferring negative information because it is easily and efficiently implemented. Details on SLD resolution, negation as failure, etc. can be found in a book by Lloyd [8].

8 Conclusion

We have presented a general logical technique for isolating the semantic effects of changes to a software system. The technique applies to structural designs containing predicates built up from our primitives and to implementations having a classical data flow semantics. The technique improves upon a straightforward information flow analysis by decomposing the usual information flow relation into three finer-grain relations, called *directed flow relations*, and by defining transitivity of information flow axiomatically in terms of the three relations. The definition of transitivity involves several mutually recursive axioms.

It is undecidable in general to determine the semantic effects of a change. Consequently, we relaxed the requirement that the results of our analysis be exact and insisted only that the results be reasonably close to exact and conservative. By relaxing the exactness constraint, we were able to use structural proofs to approximate the true semantic effects of a change. This led to a decision procedure for approximating the effects of changes, which we believe is an important step in making a formal analysis of changes practical.

However, the direct implementation of our technique in Prolog proved inefficient for systems containing a large number of objects. Further research is needed to develop a fast algorithm for computing the transitive closure of the information flow relation from our transitivity axioms.

Acknowledgements

José Meseguer, David Notkin, and John Rushby provided helpful comments on an earlier draft. Friedrich von Henke additionally suggested the formalization of context that appears in this paper.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] E. Cohen. Information transmission in computational systems. In *Proceedings of the Sixth ACM Symposium on Operating System Principles*, pages 133–139, November 1977.

[3] K.D. Cooper, K. Kennedy, and L. Torczon. The impact of interprocedural analysis and optimization in the $\mathbf{R}^n$ programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):491–523, October 1986.

[4] L.D. Fosdick and L.J. Osterweil. Data flow analysis in software reliability. *Computing Surveys*, 8(3):305–330, September 1976.

[5] R. Hood, K. Kennedy, and H.A. Müller. Efficient recompilation of module interfaces in a software development environment. In *Proceedings of ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 180–189, Palo Alto, California, December 1986.

[6] S. Horwitz, J. Prins, and T. Reps. Integrating non-interfering versions of programs. In *Proceedings of Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 133–145, San Diego, California, January 1988.

[7] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of ACM SIGPLAN 88 Conference on Programming Language Design and Implementation*, pages 35–46, Atlanta, Georgia, June 1988.

[8] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1984.

[9] J. Martin and McClure C. *Diagramming Techniques for Analysts and Programmers.* Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[10] L.M. Masinter. *Global program analysis in an interactive environment.* Technical Report SSL-80-1, Xerox Palo Alto Research Center, Palo Alto, California 94304, January 1980.

[11] M. Moriconi. A designer/verifier's assistant. *IEEE Transactions on Software Engineering*, SE-5(4):387–401, July 1979. Reprinted in *Artificial Intelligence and Software Engineering*, edited by C. Rich and R. Waters, Morgan Kaufmann Publishers, Inc., 1986. Also reprinted in *Tutorial on Software Maintenance*, edited by G. Parikh and N. Zvegintzov, IEEE Computer Society Press, 1983.

[12] M. Moriconi and D.F. Hare. The PegaSys system: Pictures as formal documentation of large programs. *ACM Transactions on Programming Languages and Systems*, 8(4):524–546, October 1986.

[13] R. Reiter. On closed world data bases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 55–76, Plenum Press, New York, New York., 1978.

[14] R. Reiter. On structuring first order data bases. In R. Perrault, editor, *Proceedings of the Canadian Society for Computational Studies of Intelligence*, pages 90–99, July 1978.

[15] C.E. Shannon. A mathematical theory of communication. *Bell Systems Technical Journal*, 27:379–423 (July), 623–656 (October), 1948.

[16] J.C. Shepherdson. Negation as failure: A comparison of Clarke's completed data base and Reiter's closed world assumption. *Journal of Logic Programming*, 1(1):51–79, June 1984.

[17] W.F. Tichy. Smart recompilation. *ACM Transactions on Programming Languages and Systems*, 8(3):273–291, July 1986.

[18] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.